Chapter Twelve

MANIPULATING DATA IN COMMAND PROCEDURES

Introduction

In the first eleven lessons we have formatted and manipulated data files and have discussed how to get information to and from the user and files. Both of these preliminary steps have been aimed at getting data or information to the command procedure so it can do our work for us. Our next logical step is to explore how we can have the command procedure do that work.

You have learned that DCL allows the OpenVMS user to represent numeric, character, or logical values with symbols. This lesson continues the discussion of symbols and symbol operations as a method of manipulating and testing data.

In addition to symbol operations, the OpenVMS operating system also provides lexical functions as a means of manipulating data. This chapter will cover several of the most useful lexical functions. For reference purposes, a list is provided at the end of this chapter which describes the majority of lexical functions and their uses.

Objectives

To use symbols and lexical functions to manipulate data in a command procedure, a programmer should be able to:

- Use symbols to compare and manipulate character strings
- Use symbols to format output
- Perform symbol substitution in command procedures
- Use appropriate lexical functions to format and manipulate data

Chapter Terms

The topics discussed in this lesson assume that the student is familiar with the following terms.

Expression	Any combination of one or more symbols, variables, or operators that have a value.
Integer symbol	A variable that can contain the integers 0 through 9.
Offset	The number of characters from the first, counting from the left, to go before performing any task.
Operator	+ (addition), - (subtraction), * (multiplication), and / (division).
String	Any combination of ASCII characters between quotation marks (“ “).
String symbol	A variable that can contain any combination of ASCII characters and is delineated by quotation marks (“ “).
Variable	A quantity that can assume any one of a set of values.

Using Symbols to Manipulate Data

String Operations

String operations can be used to join strings together or remove characters from strings. The result of a string operation is a character string value.

You can use symbol operations to manipulate data in a command procedure in a number of ways, such as:

- Creating a new record or writing a report record by taking a single word from a data record and combining it with a word or words from another data record.
- Creating a new character string by removing or replacing parts of words.
- Comparing strings to test whether the data is what you expected; based on that comparison, you may choose whether you would like to complete the task or stop the command procedure.
- Performing simple math operations.

Manipulating Strings

You can use these operators to perform string operations:

+ **String Concatenation**
  Concatenates two character strings to form a single character string.

- **String Reduction**
  Subtracts one character string from another.

Example	Result
A = “MYFILE” + “.MEM”	“MYFILE.MEM”
B = “FILENAME.MEM” - “FILE”	“NAME.MEM”
C = “LISTING.LIS” - “LIS”	“TING.LIS”
D = “MIS” + C	“MISTING.LIS”

The following rules apply to string manipulation:

- For string concatenation or reduction to occur, all operands must be character string expressions. Otherwise, any string will be converted to an integer and the result will be an integer.

```
$ X = “TESTING” + 17
$ SHOW SYMBOL X
  X = 18  Hex = 00000012 Octal = 00000000022
```

X = 18 because “TESTING” starts with the letter “T”, which is equivalent to “TRUE”, which has a value of 1. Therefore 1 + 17 = 18.

- If the string following the minus sign in a string reduction operation occurs more than once in the preceding string, only the first occurrence is removed.

```
$ Y = “TESTING” - “T”
$ SHOW SYMBOL Y
  Y = “ESTING”
```

- If you wish to remove both “T”’s:

```
$ Y = “TESTING - “T” - “T”
$ SHOW SYMBOL Y
  Y = “ESING”
```

Manipulating Arithmetic Expressions

Rules for Arithmetic Operations

- All nondecimal values (specified by radix operators) are converted to integer values before the operation is performed.
- All arithmetic is integer arithmetic.
  - Fractional values are truncated.
  - Integer arithmetic examples:

Expression	Result
5 + 4	9
-5 + 4	-1
6 / 3	2
6 / 4	1
%X10 + 5	21

Fractional arithmetic is not possible with symbols.

An arithmetic expression can contain:

- Integers
- Lexical functions that evaluate to integers
- Symbols that have integer values
- Integer operands that are connected by arithmetic, logical, and comparison operators

Operator	Meaning
+	Arithmetic sum (addition)
-	Arithmetic difference (subtraction)
+	Arithmetic unary plus (positive)
-	Arithmetic unary negate (negative)
*	Arithmetic product (multiplication)
/	Arithmetic division (integer quotient)

The following rules apply to arithmetic operations:

- The result of an arithmetic operation is an integer.
- All results which are decimal fractions are truncated.

The truncation of fractional results is illustrated in the following table:

Calculation	Actual	Result
99/100	0.99	0
101/100	1.01	1
199/100	1.99	1

Using Lexical Functions to Manipulate Data

Lexical Functions

Lexical functions are command language constructs that can perform complex operations related to files, processes, and other OpenVMS entities. They can compute all kinds of values; return the results of character string manipulations; and return information about the current process, system time, or the name of the local node on the network.

Each lexical function has a name beginning with F$ and includes a meaningful word or phrase that describes what the function does. Arguments to the lexical function follow the name inside a set of parentheses. The arguments are position dependent and are separated from each other by commas. There are a few lexical functions which require no arguments, but the parentheses must still be included.

Lexical functions return integers or character strings depending on their particular actions. If the result is a character string, it will be returned in uppercase.

Character manipulation lexical functions allow you to obtain and manipulate character strings. Some of the operations you can perform include:

- Returning character strings in a specified format
- Extracting specific parts of a character string
- Editing a string that is returned
- Returning the integer equivalent of a specified string
- Returning the length of a string
- Locating a substring within a string and returning the offset position.

Format and Syntax of Lexical Functions

All lexical functions begin with F$, and are followed by the function name. When used in the function format, the result of the function is assigned to a symbol. In the following example, the lexical function F$DIRECTORY returns the current directory name to the symbol DIR_NAME:

```
DIR_NAME = F$DIRECTORY()
```

All lexical functions have arguments enclosed in parentheses. Multiple arguments are separated by commas. When arguments are optional they may be omitted, but they are still represented by commas in the argument list. In lexical functions that require no arguments, the parentheses must still be included. The following are the types of arguments found in lexical functions and examples of each:

- Integer or character strings as a literal value.

```
WHAT = F$EXTRACT(0,3,"MAILMAN")
```

- Symbols

```
HOWLONG = F$LENGTH(X)
```

- Keywords

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

- Null arguments

```
WHEN = F$TIME()
```

- A nested lexical function

```
HowBig = F$LENGTH (F$DIRECTORY())
```

- A lexical function in an arithmetic expression

```
FileSize = F$LENGTH (F$DIRECTORY() + 10)
```

- A lexical function in a character expression

```
Date = "The date is: " +
            F$EXTRACT (0,11,F$TIME())
Time = "The time is: " +
            F$EXTRACT (12,12,F$TIME())
```

Some Useful Lexical Functions

F$CVTIME

F$CVTIME returns information about absolute, combination, or delta time strings. The syntax of this lexical function is:

```
F$CVTIME ([input-time][,output-time-format]
    [,output-field])
```

Some of the arguments for this lexical function include:

Output-time-format:

- ABSOLUTE — The requested information should be returned in military time format.
- DELTA — TODAY, YESTERDAY, TOMORROW
- COMPARISON — The requested information should be returned in the form: **"yyyy-mm-dd hh:mm:ss.cc"** (This is the default argument.)

Output-field:

- DATE — The date field is returned.
- DATETIME — The entire date and time string is returned (default).
- DAY — The day field is returned.
- HOUR — The hour field is returned.
- MONTH — The month field is returned. This selection cannot be combined with the DELTA format.
- TIME — The time field is returned.
- WEEKDAY — The weekday that corresponds to the input time argument is returned. This selection cannot be combined with the DELTA argument.
- YEAR — The year field is returned. This selection cannot be combined with the DELTA argument.

```
$ DAY = F$CVTIME("TODAY",,"WEEKDAY")
$ SHOW SYMBOL DAY
  DAY = "Thursday"
$
$ NEXT = F$CVTIME("TOMORROW",,"WEEKDAY")
$ SHOW SYMBOL NEXT
  NEXT = "Friday"
$
```

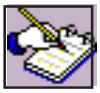
Example 12-1 - Using F$CVTIME

You can omit optional arguments to the right of the last argument you specify. The following example includes omitted optional arguments and the use of the TIME lexical function.

```
$ TIME = F$TIME()
$ SHOW SYMBOL TIME
  TIME = "17-JUN-2001 15:51:41.26"
$ TIME = F$CVTIME(TIME)
$ SHOW SYMBOL TIME
  TIME = "2001-06-17 15:51:41.26"
$
```

Example 12-2 - Using F$CVTIME With Omitted Arguments

Notes: The F$CVTIME examples



- If the input-time argument is omitted or is a null string (""), the current system date and time (in absolute format) is used.
- If the input-time argument is a delta time and the output-time-format argument is DELTA, you cannot specify MONTH, WEEKDAY, or YEAR for the output-field.
- When the weekday is returned, the first letter is in uppercase and the rest are in lowercase.
- Other values for the output-time-field include:
  - MINUTE — The minute field is returned.
  - SECOND — The second field is returned.
  - HUNDREDTH — The hundredth of a second or centisecond field is returned.

F$EDIT

F$EDIT edits a string expression. The syntax for of the F$EDIT lexical function is:

F$EDIT(*string,edit-list*)

Values you can supply for the edit-list argument include:

- COLLAPSE — Removes all spaces and tabs from the string.
- COMPRESS — Replaces multiple spaces and tabs with a single space.
- LOWERCASE — Makes the entire string lowercase.
- TRIM — Removes leading and trailing spaces and tabs from the string.
- UNCOMMENT — Removes comments from the string. (The comment portion of a string is defined as the exclamation point and all characters following it.)
- UPCASE — Makes the entire string uppercase.

The following example demonstrates the use of the F$EDIT lexical function with the COMPRESS and UPCASE edit formats. As you can see, multiple edit list items can be combined by surrounding the items in quotes and separating them with commas:

```
$ LONG = "  this  is   in lowercase with extra spaces  in  it   "
$  SHORT  =  F$EDIT(LONG,"COMPRESS,UPCASE")
$  SHOW  SYMBOL  SHORT
   SHORT  =  "THIS IS IN LOWERCASE WITH EXTRA SPACES IN IT"
$
```

Example 12-3 - Using F$EDIT With COMPRESS and UPCASE

F$EXTRACT

The F$EXTRACT lexical function extracts a substring from a character string expression. The syntax of the function is:

F$EXTRACT(*start,length,string*)

In the following example, note that the extraction starts with the first character in the supplied string if the offset is 0.

```
$ FULL_NAME = JUSTIN THYME"
$ FIRST_NAME = F$EXTRACT(0,6,FULL_NAME)
$ SHOW SYMBOL FIRST_NAME
  FIRST_NAME = "JUSTIN"
$
```

Example 12-4 - Using the F$EXTRACT Function

F$LENGTH

The F$LENGTH lexical function returns the length of the supplied string. The syntax of the function is:

F$LENGTH(*string*)

In the following example of the F$LENGTH function, note that the supplied string argument must be surrounded in quotes (unless it is a symbol name).

```
$ LEN = F$LENGTH("HOW LONG IS THIS STRING?")
$ SHOW SYMBOL LEN
  LEN = 24  Hex = 00000018  Octal =
00000000030
$
```

Example 12-5 - Using the F$LENGTH Function

F$LOCATE

The F$LOCATE lexical function locates a substring in the supplied string and returns the offset position. The following is the syntax of the function:

F$LOCATE (*substring,string*)

The example that follows demonstrates the use of the F$LOCATE function:

```
$ LONG = "This is a long string.  We are looking for the word 'long'."
$ FIND = F$LOCATE("long",LONG)
$ SHOW SYMBOL FIND
  FIND = 10  Hex = 0000000A  Octal = 00000000012
$
```

Example 12-6 - Using the F$LOCATE Function

Notes: Using the F$LOCATE Function

- The returned value is the offset of the substring argument.
- If the substring is found, this value ranges from 0 to the length of the searched string minus one.
- If the substring is not found, F$LOCATE returns the length of the searched string.

F$ SEARCH

In the OpenVMS operating system, file I/O and I/O support are managed by the ***Record Management Services (RMS)***. The F$SEARCH lexical function invokes the RMS $SEARCH routine to search a directory file and to return the full file specification of a file you name. The syntax for the F$SEARCH function is:

F$SEARCH (*file-specification,[stream-id]*)

The following information will help you better understand and use the ***stream-id*** argument:

- The search stream identification number is used to maintain separate search contexts when you use the F$SEARCH function more than once and when you supply different *file-specification* arguments.
- If you use F$SEARCH more than once in a command procedure, and if you also use different file-specification arguments, specify stream-id arguments to identify each search separately.
- If you omit the stream-id, F$SEARCH assumes an implicit single search stream. That is, it starts searching at the beginning of the directory file each time you specify a different file-specification argument.

The following example demonstrates the use of the F$SEARCH lexical function. Note that when found, a file-specification is returned as a character string. If no files meet the file-specification argument, a null string ("") is returned:

```
$ IS_IT_THERE = F$SEARCH ("TEST.COM",)
$ SHOW SYMBOL IS_IT_THERE
  IS_IT_THERE = "FAL$DISK_USER:[STUDENT07]TEST.COM;5"
$ IS_IT_THERE = F$SEARCH ("NO.FILE",)
$ SHOW SYMBOL NO.FILE
  IS_IT_THERE = ""
$
```

Example 12-7 - Using the F$SEARCH Function

When using the F$SEARCH, a number of situations may occur during the search. The following will aid you in obtaining the full set of information required by your set of searches:

The ***file-specification*** argument:

- The F$SEARCH function does not supply defaults for the file name or file type, it will accept the system defaults for your current disk and/or directory. If the version number is omitted, the specification with the highest version number is returned.

- If the file-specification argument contains wildcards, each time F$SEARCH is called the next file specification that agrees with the file-specification argument is returned (assuming you are using the stream-id). If there are no wildcards, the same file may be returned.

- A null string is returned after the last file specification that agrees with the file-specification argument.

The ***stream-id*** argument:

- The search stream identification number is used to maintain separate search contexts when you use the F$SEARCH function more than once and when you supply different file-specification arguments.

- If you use F$SEARCH more than once in a command procedure, and if you also use different file-specification arguments, specify stream-id arguments to identify each search separately.

- If you omit the stream-id, F$SEARCH assumes an implicit single stream search. That is, it starts searching from the beginning of the directory file each time you specify a different file-specification argument.

F$TIME

The F$TIME lexical function returns the current date and time string. The syntax for this function is:

F$TIME ()

The return string has the following fixed-length, 23-character format:

dd-mmm-yyyy hh:mm:ss.cc

When the current day of the month is any of the values from 1 through 9, the first character in the returned string is a blank character.

The time portion of the returned string is always in character position 13, at an offset of 12 characters from the beginning of the string

The following is an example of the F$TIME lexical function in use:

```
$ NOW = F$TIME()
$ SHOW SYMBOL NOW
  NOW = "17-NOV-2001 16:11:15.35"
$
```

Example 12-8 - Using the F$TIME Function

Using Symbols to Format Output

Now that you have performed some tasks on a character string, you will want to be able to write the new character string out to a file or record.

Creating Character String Symbols

You have already created character string symbols by using an equal sign and then enclosing the character string in quotes, as in the following symbol assignment:

```
X = "THIS IS A SYMBOL"
```

When using symbol overlays to format data, you will need to use a different assignment format. You will use:

:= to create a local character string symbol, and

:== to create a global character string symbol.

When creating a symbol with the **:=** or the **:==** signs, do not enclose the character string in quotation marks. The *colon-equal(s)* tells the OpenVMS operating system that what follows is a character string value.

The following example demonstrates three uses of the the creation of local and global character string symbols. Note that in the third case, the the equal sign is used to preserve the string case (rather than the colon-equal(s) combination).

```
$ X := Marsha Mello
$ SHOW SYMBOL X
  X = "MARSHA MELLO"
$ Y :== Marsha Mello
$ SHOW SYMBOL Y
  Y = "MARSHA MELLO"
$ Z = "Marsha Mello"
$ SHOW SYMBOL Z
  Z = "Marsha Mello"
```

Example 12-9 - Using the Local and Global Character String Symbols

String Overlays

String overlays can be used to create output records with aligned columns. The syntax for performing local and global substring overlays is:

```
$ symbol-name[offset,size] := string
$ symbol-name[offset,size] :== string
```

- The square brackets surrounding the offset and size are required notation. No spaces are allowed between the symbol name and the left bracket.
- The ***symbol-name*** can be undefined initially. The assignment statement creates the symbol name and, if necessary, provides leading or trailing spaces in the symbol value.
- The ***offset*** is an integer that indicates the position of the replacement string relative to the first character in the original string. An offset of 0 refers to the first character in the symbol, an offset of 1 refers to the second character, etc. Integer values can range from 0 through 768.
- The ***size*** is an integer that indicates the length of the replacement string.
- The ***string*** must be a character string.

The format of the symbol assignments in the following example will not result in column alignment:

```
$ NAME    := Amanda Lynn
$ PHONE   := 888-8888
$ WRITE SYS$OUTPUT NAME,PHONE
AMANDA LYNN888-8888
$
```

Example 12-10 - Using String Overlays with No Alignment

The following example illustrates how a substring overlay can achieve aligned columns:

```
$ RECORD[0,20] := 'NAME'
$ RECORD[23,11] := 'PHONE'
$ WRITE SYS$OUTPUT RECORD
AMANDA LYNN           888-8888
$
```

Example 12-11 - Using String Overlays that Achieve Alignment

Arithmetic Overlays

Arithmetic overlays can be used to create integer expressions, define binary overlays, or equate a symbol to a non-printable character (such as the ringing of a bell). The syntax for performing local and global substring overlays is:

```
$ symbol-name[bit-position,size] := expression
$ symbol-name[bit-position,size] :== expression
```

- The square brackets surrounding the offset and size are required notation. No spaces are allowed between the symbol name and the left bracket. Literal values are assumed to be decimal.
- When the ***symbol-name*** you specify is either undefined initially or defined as a string, the result of the overlay is a string. Otherwise, the result is an integer.
- The ***bit-position*** is an integer that indicates the bit position of the replacement expression in the integer. An offset of 0 refers to the first bit position in the integer, an offset of 1 refers to the second bit position, etc. Integer values can range from 0 through 32.
- The ***size*** is an integer that indicates the length of the replacement expression. Specify a size of 32 to take up the entire integer. (The maximum length is 32 in a numeric overlay and 64 in a binary overlay.)
- The ***expression*** must be a numeric expression.
- A special form of the arithmetic assignment statement can be used to perform ***binary overlays*** of the symbol value. In this case the bit-position is an integer that indicates the overlay location relative to bit 0. The size is an integer that indicates the number of bits to be overlaid.
- Binary overlays can also be used to equate a symbol to ring a bell. In this case, the bell is generated by the Ctrl/G key combination, which has the ASCII value of %X07. The following statements show how this would be achieved:

```
$ BELL[0,64] = %X07
$ WRITE SYS$OUTPUT BELL
```

Using Symbols to Test Data

In addition to being able to manipulate and format data, you can use symbols for testing data in command procedures. The ability to test for certain values and validate variable conditions helps to determine what tasks need to be performed in a command procedure.

Another use for symbols is to support ***looping structures***. You can test symbols in a looping procedure to determine the next step.

Comparing Strings

String comparison operators are available for use when testing string validity. They are similar to integer comparison operators with the following exception: the letter "S" is added to the end of the operator name. If the "S" is omitted, and the integer comparison operator is used (such as '.EQ.' as opposed to '.EQS.'), the strings or string symbols are converted to integers for the comparison. String comparisons are based on the binary values of the ASCII characters in the string. The table below provides a list of the string comparison operators.

Operator	Meaning
.EQS.	String equal to
.GES.	String greater than or equal to
.GTS.	String greater than
.LES.	String less than or equal to
.LTS.	String less than
.NES.	String not equal to

The following rules apply to string comparison:

- The comparison is on a character-by-character basis, and terminates when two characters do not match.
- If one string is longer than the other, the shorter string is padded on the right with nulls (an ASCII value of %O00) before the comparison is made; a null has a lower numeric value than any of the alphabetic or numeric characters.
- Comparisons are case-sensitive; lowercase letters have higher numeric values (beginning at %O141 / Decimal 97) than uppercase letters (beginning at %O101 / Decimal 65).
- If the result of a comparison is true, the expression is given a value of 1.
- If the comparison is false, the expression is given a value of 0.

Operands in a string comparison are string expressions. If you specify an integer value as an operand, it is converted to a string before the comparison is performed. The following table provides some character string comparisons, with their results and explanations:

Comparison Expression	Result	Explanation
“MAYBE” .LTS. “maybe”	1 (true)	The expression is true because the ASCII value of “M” is less than that of “m”.
“ABCD” .LTS. “EFG”	1 (true)	The expression is true because the ASCII value of “A” is less than that of “E”.
“YES” .GTS. “YESS”	0 (false)	The expression is false because the ASCII value of a null character is less than that of “S”. (The “YES” was padded on the right with a null.)
“AAB” .GTS. “A”	1 (true)	The expression is true because the ASCII value of “B” is greater than that of “A”.
“TRUE” .EQS. 1	0 (false)	DCL converts the integer 1 to the string “1” before comparing the ASCII value of “T” to the ASCII value or “1”.
“FALSE” .EQS. 0	0 (false)	DCL converts the integer 0 to the string “0” before making the comparison.
“123” .EQS. 123	1 (true)	DCL converts the integer 123 to the string “123” before making the comparison.

Comparing Arithmetic Expressions

Arithmetic comparison operators are used to compare integers or integer expressions. These arithmetic operators do not end in "S" because they are not used to compare strings. The result of an arithmetic comparison is an integer. The following table provides a list of the arithmetic comparison operators.

Operator	Meaning:
.EQ.	Arithmetic equal to
.GE.	Arithmetic greater than or equal to
.GT.	Arithmetic greater than
.LE.	Arithmetic less than or equal to
.LT.	Arithmetic less than
.NE.	Arithmetic not equal to

The following rules apply to arithmetic comparisons:

- Operands in arithmetic comparisons are integer expressions.
- If you specify a character string value as an operand, the string is converted to an integer value before the comparison is performed.
- If a character string begins with an upper- or lowercase "T" or "Y", the string is converted to the integer 1. If the string begins with any other letter, the string is converted to the integer 0.
- If a string contains numbers that form a valid integer, the string is converted to its integer equivalent.
- If the result of an arithmetic comparison is true, the expression has a value of 1. If the result of the comparison is false, the expression has a value of 0.

The following table provides some examples of arithmetic comparisons.

Expression	Value of Expression	Explanation
1 .LE. 2	1 (true)	The expression is true because the integer 1 is less than the integer 2.
1 .GT. 2	0 (false)	The expression is false because the integer 1 is not greater than the integer 2.
1 + 3 .EQ. 2 + 5	0 (false)	The expression is false because the integer 4 (the sum of 1 + 3) is not equal to the integer 7(the sum of 2 + 5).
“TRUE” .EQ. 1	1 (true)	The expression is true because the string “TRUE” is converted to the integer 1 for comparison with the integer 1, and they are equal.
“FALSE” .EQ. 0	1 (true)	The expression is true because the string “FALSE” is converted to integer 0 for the comparison with the integer 0, and they are equal.
“123” .EQ. 123	1 (true)	The expression is true because the string “123” is converted to the integer equivalent for comparison with the integer 123, and they are equal.

Lexical Function Reference List

This reference list is a partial set of lexical functions available to users when seeking information or manipulating data. The functions that are described on the following pages address the needs for:

- Process information
- System information
- Character manipulation
- File information

Process Information Lexicals

Process information lexicals are lexical functions used to obtain data about processes. They allow you to save and restore the characteristics of your process.

F$DIRECTORY

The F$DIRECTORY function returns the current default directory as a character string. The function does not return the associated device. The syntax of the function is:

```
F$DIRECTORY ()
```

F$ENVIRONMENT

The F$ENVIRONMENT function returns information about the DCL command environment. The syntax of the function is:

```
F$ENVIRONMENT (item)
```

Some of the default items you can supply to this lexical function are:

- DEFAULT — Returns the default device and directory name as a character string.
- KEY_STATE — Returns a character string indicating the current locked keypad state.
- MESSAGE — Returns a character string containing the current setting of the SET MESSAGE command.
- PROCEDURE — Returns the file specification for the command procedure from which the F$ENVIRONMENT("PROCEDURE") function is issued. The file specification is returned as a character string. If the lexical function is invoked interactively, a null string ("") is returned.
- PROMPT — Returns a character string containing the current prompt string.
- PROTECTION — Returns a character string indicating the current default file protection.

F$GETJPI

The F$GETJPI function invokes the $GETJPI system service to return process information about the specified process. The syntax of the F$GETJPI function is:

F$GETJPI (*process-id,item*)

The ***process-id (PID)*** specifies the identification number of the process for which the information is being reported. If the process-id is omitted or is a null string (""), the current process identification number is used. You cannot use a wildcard for the PID argument. To get a list of PID's use the F$PID lexical function.

The ***item*** argument must be specified as an uppercase character string expression. Use online help to view some of the items you can supply to this lexical function.

F$MODE

The F$MODE lexical function returns a character string indicating the mode that a process is running (e.g., INTERACTIVE, NETWORK, BATCH, or OTHER). The syntax of this function is:

F$MODE ()

F$PROCESS

The F$PROCESS lexical function returns the current process name. There are no arguments required for this function. The syntax of this function is:

F$PROCESS ()

F$USER

The F$USER lexical function returns the current User Identification Code (UIC) in the group/member name format (as opposed to the number format). The UIC returned is a character string containing the current UIC, including the square brackets. The syntax of this function is:

F$USER ()

System Information Lexicals

System information lexicals allow the user to obtain the following types of information about system parameters: system message text; logical names and their attributes; and date and time.

F$GETSYI

The F$GETSYI lexical function invokes the $GETSYI system service to return status and identification information about your system or a node in your cluster. The syntax of the F$GETSYI function is:

```
F$GETSYI  (item[,node-name][,cluster-id])
```

Items must be specified as upper case character string expressions. Returned values are either integers or character strings, depending upon your request.

F$MESSAGE

The F$MESSAGE lexical function returns the message text associated with a specific system status code. The syntax of the F$MESSAGE function is:

```
F$MESSAGE  (status-code[,message-component-list])
```

The ***status-code*** is specified as an integer expression.
The ***message-component-list*** specifies the system message component for which the information is to be returned. Valid message component keywords are:

- FACILITY - Facility name
- SEVERITY - Severity level indicator
- INDENT - Abbreviation of message text
- TEXT - Explanation of message

If you specify an argument that has no corresponding message, F$MESSAGE returns a string containing the NOMSG error message.

F$TIME

The F$TIME lexical function returns the current date and time string. The syntax of this function is:

```
F$TIME  ()
```

The returned string has the following fixed, 23-character format:

```
dd-mmm-yyyy  hh:mm:ss.cc
```

F$TRNLNM

The F$TRNLNM lexical function returns the equivalence string or requested attributes associated with the logical name. The function does not perform iterative translation. The syntax of the F$TRNLNM function is:

```
F$TRNLNM (logical-
name[,table][,index][,mode][,case][,item])
```

The return value is either an integer or a character string, depending upon the ***item*** you request. If no match is found, a null string is returned.

The ***table*** argument defaults to LNM$DCL_LOGICAL (generally the process, job, group, and system tables - in that order.)

Some of the values you can supply for the item argument include:

- CONCEALED — Returns one of the character strings “TRUE” or “FALSE” to indicate whether the CONCEALED attribute was specified.
- CONFINE — Returns one of the character strings “TRUE” or “FALSE” to indicate whether the logical name is confined. If the logical name is confined (true), then the name will not be copied to subprocesses.
- TABLE — Returns one of the character strings “TRUE” or “FALSE” to indicate whether the logical name is the name of a logical name table.
- TABLE_NAME — Returns the name of the name table where the logical name was found.
- TERMINAL — Returns one of the character strings “TRUE” or “FALSE” to indicate whether the TERMINAL attribute was specified.
- VALUE — Returns the equivalence name associated with the specified logical name.

Character Manipulation Lexicals

Character manipulation lexicals are functions that allow you to obtain and operate on character strings. The following is a list of some of the operations you can perform:

- Return character strings to yourself in a specified format
- Extract specific parts of a character string
- Edit a string that is returned to you
- Convert character and numeric input to character strings
- Return the integer equivalent of a specified string
- Return the length of a string
- Locate a substring within a string and return the offset position

The lexical functions in this section provide character manipulation operations for your use.

F$FAO

The F$FAO lexical function converts character and numeric input to ASCII character strings (FAO stands for Formatted ASCII Output). By specifying formatting instructions, you can use the F$FAO function to convert integer values to character strings, to insert carriage returns and form feeds, to insert text, set up tabular columns, etc. The syntax of the F$FAO function is:

F$FAO (*control-string [,argument[,...]]*)

In the F$FAO function the ***control-string*** field specifies the fixed text of the output string. The *control-string* may be of any length and is specified as a character string expression.

In this function the ***argument*** field specifies from 1 to 15 arguments required by F$FAO directives used in the control string. Each argument string can describe an output field. The order of the arguments must correspond exactly with the order of the directives in the control string. If an argument is misplaced, no error message will be displayed, in most cases. If you specify an argument whose type does not match that of the corresponding control string directive, unpredictable results are returned.

F$INTEGER

The F$INTEGER lexical function returns an integer equivalent of the specified string. The format of the function is:

F$INTEGER (*string*)

If you supply an integer expression in the ***string*** argument, the F$INTEGER function evaluates the expression and returns the result. If you specify a string expression, the function converts the string to an integer before returning the result.

F$STRING

The F$INTEGER lexical function returns a character string that is equivalent to the specified expression. The format of the function is:

F$STRING (*expression*)

The expression supplied as the F$STRING argument can be an integer, an integer expression, or a string expression. If you supply an integer expression, the F$STRING function converts the resulting integer to a string, and returns the result. If you specify a string expression, the F$STRING functions returns the string result.

When converting an integer or integer expression to a string, the F$STRING function uses decimal representation and omits leading zeros. When converting a negative integer, the function places a minus sign at the beginning of the string result.

File Information Lexicals

File information lexicals allow you to:

- Return attribute information for a specific file in integer or character form
- Parse a file specification and return all or any part of it
- Search a directory file and return a file specification

The lexical functions in this section provide file information operations for your use.

F$FILE_ATTRIBUTES

The F$FILE_ATTRIBUTES lexical function returns attribute information for a specific file in either integer or character string form, depending on the item you request. The syntax of the lexical function is:

F$FILE_ATTRIBUTES (*file-specification, item*)

The ***file-specification*** argument specifies the name of the file about which you are requesting information in the form of a character string. Only one file can be specified and wildcards are not allowed.

The ***item*** argument indicates which attribute of the file is to be returned. The item must be specified as a character string expression. Some of the values you can supply for the item argument include:

- ALQ — Returns Allocation quantity
- **BDT** — **Returns Backup date/time**
- BKS — Returns Bucket size
- **BLS** — **Returns Block size**
- CBT — Contiguous-best-try (True/False)
- **CDT** — **Returns the creation date/time string**
- CTG — Contiguous (True/False)
- **DID** — **Returns Directory ID string**
- DIRECTORY — Returns TRUE if it is a directory
- **DVI** — **Returns Device name string**
- EDT — Returns Expiration date/time
- **EOF** — **Returns the number of blocks used (integer).**

- ERASE — Returns TRUE if file's contents are erased before file is deleted
- **FID** — **Returns the File ID string**
- GBC — Returns the global buffer count
- **GRP** — **Returns Owner group number**
- KNOWN — Means known file. This item returns the value "TRUE" or "FALSE" to indicate whether the file is installed with the OpenVMS Install utility
- **LOCKED** — **True if a file is deaccessed-locked**
- LRL — Returns Longest record length
- **MBM** — **Returns Owner member number**
- MRN — Returns Maximum record number
- **MRS** — **Returns Maximum record size**
- NOK — Returns Number of keys
- **ORG** — **Means file organization. This item returns the value "SEQ" for sequential file, "REL" for relative file, or "IDX" for indexed sequential file**
- PRO — Returns the file protection string
- **RAT** — **Returns Record attribute string**
- RCK — Returns TRUE if read-check
- **RDT** — **Returns Revision date/time**
- RFM — Record format string; possible return values are
- VAR, FIX, VFC, UDF, STM, STMLF, or STMCR
- **RVN** — **Returns Revision number**
- UIC — Returns the owner UIC string
- **WCK** — **Returns TRUE if write-check**

F$PARSE

The F$PARSE lexical function invokes the $PARSE RMS (record management services) routine to parse a file specification and to return either the expanded file specification or the particular file specification field you request. The syntax of this lexical function is:

F$PARSE (*file-specification[,default-spec][,related-spec][,field][,parse-type]*)

The ***file-specification*** argument can contain wildcard characters. If you use a wildcard character, the file specification returned will contain the wildcard.

If an error is detected during the parse, the F$PARSE function returns a null string, except when you specify a field name or the SYNTAX_ONLY parse type.

If a particular field in the file specification argument is missing, the fields in the default-file specification (***default-spec***) argument are substituted in the output string . You can make additional substitutions in the file specification argument by using the related-spec argument.

The fields in the related file specification (***related-spec***) argument are substituted in the output string if a particular field is missing from both the file-spec and default-spec arguments.

Specifying the ***field*** argument causes the F$PARSE function to return a specific portion of a file specification. When using the field argument, do not abbreviate field names. Some of the values you can supply for the field argument include:

- NODE — Node name
- DEVICE — Device name
- DIRECTORY — Directory name
- NAME — File name
- TYPE — File type
- VERSION — File version number

F$ SEARCH

In the OpenVMS operating system, file I/O and its support are managed by ***RMS, the Record Management Services***. The F$SEARCH lexical function invokes the RMS $SEARCH routine to search a directory file and to return the full file specification of a file you name. The syntax for the F$SEARCH function is:

F$SEARCH (*file-specification,[stream-id]*)

The following information will help you better understand and use the stream-id argument:

- The search stream identification number is used to maintain separate search contexts when you use the F$SEARCH function more than once and when you supply different ***file-specification*** arguments.
- If you use F$SEARCH more than once in a command procedure, and if you also use different ***file-specification*** arguments, specify stream-id arguments to identify each search separately.
- If you omit the stream-id, F$SEARCH assumes an implicit single search stream. That is, it starts searching at the beginning of the directory file each time you specify a different ***file-specification*** argument.

The F$SEARCH function returns a character string containing the expanded file specification for the file-specification argument. If the file is not found, the function returns the null string (“”).

If the device or directory names are omitted, your current default disk and/or directory are used. The F$SEARCH function does not supply defaults for file name or file type. If the version number is omitted, the specification for the file with the highest version number is returned.

If the file-specification contains wildcards, each time F$SEARCH is called, the next file specification that agrees with the file-spec argument is returned. A null string is returned after the last file specification that agrees with the file-spec argument.

Summary

Concepts

- Techniques used to manipulate symbols include string manipulation and arithmetic manipulation.

- The general format for any lexical function is:
  F$function-name([argument,...])

- Arguments to a lexical function can be:
  - Integer or character strings
  - Symbols
  - Keywords
  - Null arguments

- Parentheses are mandatory in a lexical function, even if no arguments are required.

- Many lexical functions return information. This information can be in string or integer form, depending upon the particular lexical function.

- You can use the technique of symbol overlay to manipulate data.

- Methods for testing data using symbols include:
  - String comparison
  - Arithmetic comparisons
  - Logical comparisons