Network Operating System / Virtual Environment

(NOS/VE)

System Command Language

(SCL)

Command Interface

External Reference Specification

Revision Record

Printing	Comments
current	Changed the default minimum size for a LIST type to zero when the type is not that of a command or function parameter.
	Added a default initial value of the empty list for variables of type LIST with a minimum size of zero.
	Reinstated the description of the $SUBSTRING function which was mistakenly removed in an earlier revision.
	Added the $COMBINE function for lists, a generalized form of the $APPLY function.
	Miscellaneous editorial corrections.
88/07/11	Minor corrections to the syntax of <type definitions> and <variable definitions>.
	Marked the ATTACH_FILE_DEFAULTS and FILE_ATTRIBUTE_DEFAULTS environment objects as for a future release.
	Keyword and data_name variables or functions may now be used within a file expression.
	Marked the $PROCEDURE, $TASK and $UTILITY path elements as “futures.”
	Added an option to the $CATALOG_CONTENTS function to return a list of full path names.
	Added an option to the $FILE_CYCLES function to return a list of full path names.
	The $APPLY function is now influenced by the context in which it is called, enabling it return, for example, a list of records, etc.
	Added the $JOB_TERMINATION_STATUS function.
	Added an option to the $UNIQUE function to produce its result in the form of a string, name or file.
	Removed the BY_NAME attribute from the PLACEMENT parameter of the CREATE_COMMAND_LIST_ENTRY command.
	For the WAIT command, restored the ability to specify time in milliseconds via an integer value. Also, corrected the default for the UNTIL parameter to ALL.
	Miscellaneous editorial corrections.
88/05/02	*Describes the command interface to SCL at release 1.4.1 of NOS/VE. Items marked with (*) are not part of that release but may be included in a later release.*
	Updated the description of the assignment statement with regard to the implicit creation of the variable and the rules for assignment of strings.
	Added a note about the importance of operand order when using relational operators.
	Updated the description of the division operator when both operands are integers.
	Added a note about the implicit conversion of its right operand to a string by the concatenation operator.
	Removed the special handling of the relational operator symbols within application value text.
	Added the $DATA_NAME, $DATE_TIME_STRING, $PROGRAM_NAME, $RANGE_OF, $FIELD and $CPU_TIME functions.
	Removed the proposed third parameter of the $FILE function.
	Removed $JOB as an alias for $LOCAL.
	Removed the ability to use $HIGH, $LOW and $NEXT as “stand alone” functions—they may only be used within a file expression.
	Added a DEFAULT_RADIX parameter to the $INTEGER function.
	Updated the description of the FOR EACH statement regarding the implicit creation of the control variable.
	Reinstated the IDENTIFIER parameter of the $STATUS function (for compatibility with previous systems).
	Added ADVANCED_USAGE to the DISPLAY_OPTIONS parameter of the DISPLAY_COMMAND_LIST_ENTRY command.
	Removed the adding of spaces around expression operators by FORMAT_SCL_PROC.
	Added the NAME_FOLDING_LEVEL parameter to the CHANGE_SCL_OPTIONS command.
	Added COMPACT_PARAMETER_DESCRIPTIONS to the DISPLAY_OPTIONS parameter of the

	DISPLAY_COMMAND_INFORMATION and DISPLAY_FUNCTION_INFORMATION commands, and replaced PARAMETER_DESCRIPTIONS with it as one of the defaults. Also added the definition of a “default variable” for the DISPLAY_OPTIONS parameter of these commands.
	Renamed DISPLAY_COMMAND_ENVIRONMENT to DISPLAY_COMMAND_STACK.
	Corrected the description of the GET_LINE command to have the same parameter names and order as did ACCEPT_LINE.
	Updated the description of the INCLUDE_COMMAND command to indicate that it can handle multiple commands.
	Added the ability to display variables created within the job to the DISPLAY_VARIABLE_LIST_ENTRY command.
	Added the ability to include a “description” for a variable on a variable library.
	Miscellaneous clarifications and editorial corrections.
88/01/12	Removed the option of omitting the word “integer” when declaring an integer type (this makes the diagnosis of an erroneous type specification more useful).
	In the section “Names of Variables” changed the prohibition of using the same name for a variable as for a procedure parameter to a caution against doing so.
	Clarified which task blocks can hold variables.
	Added a note that variables declared with the XREF scope may not be given an initial value.
	Clarified comparisons involving names, keywords, etc. Removed the abaility to compare “application values.” Also, all relational operations described in the ERS will be supported by the initial release of the “New Types” feature.
	Removed the “OR” (alternation) operator from string pattern expressions. When string patterns are implemented this operation will be provided by a function.
	Added an example for the $apply function.
	Changed the name of the $list function to $list_of to avoid confusion with the “standard file” $list.
	Added keywords to the “order” parameter of the $sort function.
	Updated the descriptions of the $type and $element_type functions to indicate that they must be passed a variable rather than an expression.
	Added functions $max_string_size and $min_string_size to the “Functions Related to Types” section. These functions return the maximum and minimum size attributes for a string type.
	Removed the $initialized function and added INITIALIZED as an option to the $variable function. Also, clarified other options of $variable.
	Added the ability to separate a statement label from its following colon may a space provided that the colon is also followed by a space.
	Miscellaneous editorial corrections.
87/10/23	Provided a full description of the DISPLAY_COMMAND_INFORMATION and DISPLAY_FUNCTION_INFORMATION commands includinga number of display options and how they will use “help modules.”
	Added a CHECK attribute for parameters.
	Added an <initial name> option for the initialization of variables analogous to the <default name> capability for parameters.
	Miscellaneous editorial corrections
87/10/06	Added a <value qualifier> for ranges (“.LOW” and “.HIGH”) to replace the use of the functions $LOW and $HIGH for range values.
	By popular demand, parameters with the SECURE attribute are once again allowed to have default values.
87/09/08	Updated the description of parameter prompting.
	Removed restrictions on writing fields of “built-in” records.
	Specified initial values for certain kinds of variables for compatibility.
	Clarified the handling of $SYSTEM within file expressions.
	The VAR attribute is no longer allowed for function parameters.
	The VAR and SECURE attributes are no longer allowed for the same command parameter.
	A parameter with the SECURE attribute may no longer have a default value.

	For compatibility reasons, the function previously documented in this ERS as $VALUE has been renamed to $PARAMETER_VALUE. This allows the old $VALUE function to work as it always has.
	For compatibility reasons, the function previously documented in this ERS as $RANGE has been renamed to $RANGE_SPECIFIED. This allows the old $RANGE function to work as it always has.
	Added documentation for the $VNAME function.

Table of Contents

Revision Record i

Table of Contents iv

1 SCL Concepts 1
- 1.1 Statement Submission 1
- 1.2 Statement Processing 1
- 1.3 Command Logging and Echoing 1
- 1.4 Commands 2
- 1.5 Functions 2
- 1.6 Control Statements 2
- 1.7 Condition Handling and Status Parameters 2
- 1.8 Parameters, Variables and Types 2
- 1.9 Files, Catalogs and Paths 2
- 1.10 Command Utilities 3
- 1.11 Command List 3
- 1.12 Block Structure and Scope 3
- 1.13 Text Collection and Substitution 3
- 1.14 Defaulting 3
- 1.15 Aliasing 4
- 1.16 Prologs and Epilogs 4
- 1.17 Message Templates 4
- 1.18 Message Levels 4
- 1.19 Interactive Usage 4
  - 1.19.1 Line Mode Dialogs 5
  - 1.19.2 Screen Mode Dialogs 6
  - 1.19.3 “Desktop Environment” Interaction 6
  - 1.19.4 Help 6
- 1.20 Help Modules 6
- 1.21 “Advanced” Commands, Functions, Parameters and Keywords 6
- 1.22 “Hidden” Commands, Functions, Parameters and Keywords 7
- 1.23 “Wild Card” Characters 7
- 1.24 “Hooks” for Extension 7

2 Fundamentals of SCL 8
- 2.1 Lines and Their Continuation 8
- 2.2 MetaLanguage 8
- 2.3 Building Blocks of SCL 9
  - 2.3.1 Comments 9
  - 2.3.2 Spaces 9
  - 2.3.3 Delimiters 10
  - 2.3.4 Names 10
    - 2.3.4.1 Use of the $ in Names 11
    - 2.3.4.2 Use of “International” Characters in Names 11
  - 2.3.5 Integers 11
  - 2.3.6 Real Numbers 12
  - 2.3.7 Strings 12
  - 2.3.8 Booleans 13

3 Input to the SCL Interpreter 14

4 Commands and Their Parameters 16
- 4.1 Use of Spaces in Parameter Values 16
- 4.2 Passing Omitted Parameters 16
- 4.3 Command With Prompting Statement 17

5 Data Types 18
- 5.1 Type Definitions 18
- 5.2 Elemental Types 18

5.2.1 Boolean Type 19
5.2.2 COBOL Name Type 19
5.2.3 Data Name Type 19
5.2.4 File Type 19
5.2.5 Integer Type 19
5.2.6 Keyword Type 20
5.2.6.1 Advanced Keywords 20
5.2.6.2 Hidden Keywords 20
5.2.7 Lock Type (*) 20
5.2.8 Name Type 20
5.2.9 Program Name Type 20
5.2.10 Real Type 21
5.2.11 Statistic Code Type 21
5.2.12 Status Code Type 21
5.2.13 String Type 21
5.2.14 String Pattern Type 22
5.3 Structured Types 22
5.3.1 Array Type 22
5.3.2 Command Reference Type 23
5.3.3 Date Time Type 23
5.3.4 Entry Point Reference Type 24
5.3.5 Line Identifier Type 24
5.3.6 List Type 25
5.3.7 Range Type 25
5.3.8 Record Type 25
5.3.9 Status Type 26
5.3.10 Time Increment Type 26
5.3.11 Time Zone Type 27
5.4 Union Type (Any) 27
5.5 Application Type 28
5.6 Type Specification Type 28

6 Variables 29
6.1 Defining Variables 29
6.2 Names of Variables 29
6.3 Initial Values for Variables 30
6.4 Read-Only Variables 30
6.5 Residence of Variables 30
6.5.1 The Variable Library List (*) 31
6.6 Accessibility of Variables 31
6.7 Lifetime of Variables 31
6.8 Scope of Procedure Variables 31
6.9 Scope of Environment Variables 32
6.9.1 Example of PUSH Scope 32
6.10 Deferred Evaluation of Variables 33
6.11 Variable References 33
6.12 Variable Control Statements 33
6.12.1 Assignment 33
6.12.2 PUSH 34
6.12.3 POP 34
6.12.4 LOCK (*) 34
6.12.5 UNLOCK (*) 35
6.13 Deleting Variables 35

7 Expressions 36
7.1 Elemental Expressions 36
7.1.1 Boolean Expression 36
7.1.1.1 Comparison of Strings 37
7.1.1.2 Comparison of Names, Keywords, Data_Names and COBOL_Names 37
7.1.1.3 Comparison of Program_Names 38
7.1.1.4 Comparison of File (Path) Names 38
7.1.1.5 Summary of Comparisons of Elemental Types 38
7.1.1.6 Comparison of Application Values 38
7.1.1.7 Comparison of Arrays 39

7.1.1.8 Comparison of Command Reference Values 39
7.1.1.9 Comparison of Date_Time Values 39
7.1.1.10 Comparison of Entry_Point_References 39
7.1.1.11 Comparison of Line_Identifiers 39
7.1.1.12 Comparison of Lists 39
7.1.1.13 Comparison of Ranges 39
7.1.1.14 Comparison of Records 39
7.1.1.15 Comparison of Status Values 39
7.1.1.16 Comparison of Time_Increment Values 39
7.1.1.17 Comparison of Time_Zone Values 39
7.1.1.18 Comparison of Types 39
7.1.2 COBOL Name Expression 40
7.1.3 Data Name Expression 40
7.1.4 File Expression 40
7.1.4.1 Generic Paths 43
7.1.5 Integer Expression 44
7.1.6 Keyword Expression 44
7.1.7 Lock Expression (*) 45
7.1.8 Name Expression 45
7.1.9 Numeric Expression 45
7.1.10 Program Name Expression 46
7.1.11 Real Expression 47
7.1.12 Statistic Code Expression 47
7.1.13 Status Code Expression 47
7.1.14 String Expression 48
7.1.15 String Pattern Expression 48
7.1.15.1 “Wild Card” Patterns 48
7.2 Structure Expressions 50
7.2.1 Array Expression 51
7.2.2 Command Reference Expression 51
7.2.3 Date Time Expression 51
7.2.3.1 Date and Time Formats 52
7.2.3.2 Month and Day Name Modules 54
7.2.3.3 Time Zone Identifier Modules 55
7.2.4 Entry Point Reference Expression 55
7.2.5 Line Identifier Expression 55
7.2.6 List Expression 56
7.2.6.1 “Wild Card” Patterns as List Generators 56
7.2.6.2 Path Traversal as a List Generator 56
7.2.7 Range Expression 57
7.2.8 Record Expression 57
7.2.9 Status Expression 57
7.2.10 Time Increment Expression 57
7.2.11 Time Zone Expression 58
7.3 Application Expression 58

8 Command and Function Procedures 60
8.1 Command Procedures 60
8.2 Function Procedures 61
8.3 Procedure Scope Attributes 62
8.4 Help Module References 62
8.5 Advanced Commands and Functions 62
8.6 Hidden Commands and Functions 63
8.7 Parameter Passing Modes 63
8.8 Deferred Evaluation of Parameters 63
8.9 Advanced Parameters 63
8.10 Hidden Parameters 64
8.11 Secure Parameters 64
8.12 The Type Name Parameter Attribute 64
8.13 Default Values for Parameters 64
8.14 Extended Parameter Checking 64
8.14.1 The CHECK Parameter Attribute 65
8.14.2 CHECK / CHECKEND (*) 65
8.15 Command Log Options 65

9 9 Funtions 67
9.1 Function Calls 67
9.2 Functions Related to Arrays 67
9.2.1 $array 67
9.2.2 $lower_bound 67
9.2.3 $upper_bound 68
9.3 Functions Related to Booleans 68
9.3.1 $and (*) 68
9.3.2 $boolean (*) 68
9.3.3 $if 68
9.3.4 $not 69
9.3.5 $or (*) 69
9.3.6 $xor (*) 69
9.4 Functions Related to COBOL Names 69
9.4.1 $cobol_name (*) 69
9.5 Functions Related to Command and Function Processing 70
9.5.1 $caller_interactive (*) 70
9.5.2 $caller_ring (*) 70
9.5.3 $command 70
9.5.4 $command_of_caller 70
9.5.5 $expected_type (*) 70
9.5.6 $interaction_style 71
9.5.7 $message_level 71
9.5.8 $natural_language 72
9.6 Functions Related to Command References 72
9.6.1 $command_reference (*) 72
9.7 Functions Related to Dates, Times, Time Increments, and Time Zones 72
9.7.1 $date 72
9.7.2 $date_time 73
9.7.3 $date_time_string 73
9.7.4 $day 74
9.7.5 $local_date_time 74
9.7.6 $now 74
9.7.7 $time 75
9.7.8 $time_zone 75
9.7.9 $time_zone_identifier ($time_zone_id) 75
9.7.10 $universal_date_time 76
9.8 Functions Related to Data Names 76
9.8.1 $data_name 76
9.9 Functions Related to Expressions 77
9.9.1 $evaluate 77
9.10 Functions Related to Files, Catalogs and Paths 77
9.10.1 $catalog_contents 77
9.10.2 $family 78
9.10.3 $file_attributes 78
9.10.4 $file_cycles 78
9.10.5 $fname 78
9.10.6 $high 79
9.10.7 $local 79
9.10.8 $low 79
9.10.9 $next 79
9.10.10 $path_elements 79
9.10.11 $procedure (*) 80
9.10.12 $system 80
9.10.13 $task (*) 80
9.10.14 $up 80
9.10.15 $user 81
9.10.16 $working_catalog ($wc) 81
9.11 Functions Related to Integers 81
9.11.1 $integer 81
9.11.2 $integer_string 81
9.11.3 $max_integer 82
9.11.4 $min_integer 82
9.11.5 $mod 82

9.12 Functions Related to Lists 82
9.12.1 $add 82
9.12.2 $apply 83
9.12.3 $build_list 84
9.12.4 $build_result 85
9.12.5 $combine 85
9.12.6 $difference 87
9.12.7 $first 87
9.12.8 $intersection 87
9.12.9 $join 87
9.12.10 $last 88
9.12.11 $list_of 88
9.12.12 $max 88
9.12.13 $max_list 88
9.12.14 $min 88
9.12.15 $nil 89
9.12.16 $rest 89
9.12.17 $reverse 89
9.12.18 $select 89
9.12.19 $select_string(s) 90
9.12.20 $select_wild_card_file(s), $select_file(s) 90
9.12.21 $select_wild_card_name(s), $select_name(s) 90
9.12.22 $select_wild_card_program_name(s) 91
9.12.23 $select_wild_card_string(s) 91
9.12.24 $size 92
9.12.25 $sort 92
9.12.26 $sublist 92
9.12.27 $subset 93
9.12.28 $sum 93
9.12.29 $union 94
9.12.30 $wild_card_file(s) ($wcf) 94
9.13 Functions Related to Locks 94
9.13.1 $locked (*) 94
9.14 Functions Related to Names 95
9.14.1 $max_name 95
9.14.2 $name 95
9.15 Functions Related to Parameters 95
9.15.1 $specified 95
9.15.2 $parameter_value 95
9.16 Functions Related to Program Names 96
9.16.1 $program_name 96
9.17 Functions Related to Ranges 96
9.17.1 $range_of 96
9.17.2 $range_specified 96
9.18 Functions Related to Real Numbers 97
9.18.1 $indefinite 97
9.18.2 $infinite 97
9.18.3 $infinity 97
9.18.4 $max_real 97
9.18.5 $min_real 98
9.18.6 $real 98
9.18.7 $real_string 98
9.19 Functions Related to Records 98
9.19.1 $field 98
9.19.2 $field_list 99
9.19.3 $record 101
9.19.4 $sort_fields 102
9.20 Functions Related to Statistic Codes 102
9.20.1 $statistic_code 102
9.20.2 $statistic_code_string 102
9.21 Functions Related to Statuses and Status Codes 102
9.21.1 $previous_status 102
9.21.2 $status 103
9.21.3 $status_code 103

9.21.4 $status_code_name ... 103
9.21.5 $status_code_string ... 103
9.21.6 $status_message ... 104
9.21.7 $status_severity ... 104
9.22 Functions Related to Strings ... 105
9.22.1 $char ... 105
9.22.2 $format_value ... 106
9.22.3 $justify ... 107
9.22.4 $max_string ... 107
9.22.5 $ord ... 107
9.22.6 $quote ... 108
9.22.7 $scan_any ... 108
9.22.8 $scan_not_any ... 108
9.22.9 $scan_string ... 109
9.22.10 $size ... 109
9.22.11 $string ... 109
9.22.12 $substring ... 110
9.22.13 $translate ... 110
9.22.14 $trim ... 111
9.23 Functions Related to String Patterns ... 112
9.23.1 $match ... 113
9.23.2 $sp_any ... 114
9.23.3 $sp_balance ... 114
9.23.4 $sp_capture ... 115
9.23.5 $sp_count ... 116
9.23.6 $sp_defer ... 117
9.23.7 $sp_fail ... 118
9.23.8 $sp_fence ... 118
9.23.9 $sp_index ... 118
9.23.10 $sp_left ... 119
9.23.11 $sp_not_any ... 120
9.23.12 $sp_null ... 120
9.23.13 $sp_or ... 121
9.23.14 $sp_repeat ... 121
9.23.15 $sp_right ... 122
9.23.16 $sp_stop ... 123
9.23.17 $sp_string ... 123
9.23.18 $sp_succeed ... 123
9.23.19 $sp_test ... 124
9.23.20 $sp_upto ... 124
9.23.21 $sp_wild_card, $sp_wc ... 125
9.24 Functions Related to Types ... 126
9.24.1 $element_type ... 126
9.24.2 $generic_type ... 126
9.24.3 $lower_value ... 127
9.24.4 $max_string_size ... 127
9.24.5 $min_string_size ... 127
9.24.6 $type ... 128
9.24.7 $upper_value ... 128
9.25 Functions Related to Variables ... 128
9.25.1 $variable ... 128
9.25.2 $vname ... 129
9.26 Miscellaneous Functions ... 129
9.26.1 $cpu_time ... 129
9.26.2 $job_termination_status ... 129
9.26.3 $mainframe ... 130
9.26.4 $processor ... 130
9.26.5 $read (*) ... 130
9.26.6 $system ... 131
9.26.7 $unique ... 131
9.26.8 $use_value ... 132

10 Structured Statements ... 133
10.1 Statement Labels ... 133
10.2 BLOCK / BLOCKEND ... 133

10.3 LOOP / LOOPEND ... 133
10.4 WHILE / WHILEND ... 133
10.5 REPEAT / UNTIL ... 134
10.6 FOR / FOREND ... 134
10.6.1 Incremental Control ... 134
10.6.2 List Control ... 134
10.7 IF / ELSEIF / ELSE / IFEND ... 135
10.8 CASE / <Case Selection> / ELSE / CASEND (*) ... 135

11 Error Handling ... 137
11.1 Status Parameters ... 137
11.2 Conditions and Condition Handlers ... 137
11.2.1 Classes of Conditions ... 137
11.2.2 Residence of Condition Handlers ... 138
11.2.3 WHEN / WHENEND ... 138
11.2.4 CANCEL ... 139
11.2.5 CAUSE ... 139
11.2.6 CONTINUE ... 139
11.2.7 Condition Descriptions ... 140
11.2.7.1 ANY_CONDITION ... 140
11.2.7.2 ANY_FAULT ... 140
11.2.7.3 COMMAND_FAULT ... 140
11.2.7.4 DISCONNECT ... 140
11.2.7.5 EXECUTION_FAULT ... 140
11.2.7.6 EXIT ... 140
11.2.7.7 LIMIT_FAULT ... 141
11.2.7.8 PAUSE ... 141
11.2.7.9 RECONNECT ... 141
11.2.7.10 TERMINATE ... 141
11.2.7.11 UNSEEN_MAIL ... 141
11.2.7.12 Other Conditions ... 141
11.2.7.13 User Defined Conditions ... 141

12 Control Statements ... 142
12.1 CYCLE ... 142
12.2 EXIT ... 142
12.2.1 Exiting a Command Procedure ... 143
12.2.2 Exiting a Function Procedure ... 143
12.2.3 Exiting a Utility ... 143
12.2.4 Exiting a Task ... 143

13 Command List ... 145
13.1 Command List Entries ... 145
13.1.1 Object Library ... 145
13.1.2 Catalog ... 145
13.1.3 Working Catalog ... 146
13.1.4 $system ... 146
13.1.5 Command Utility ... 146
13.1.6 Control Statements ... 146
13.1.7 Control Commands ... 146
13.1.8 Fence ... 147
13.2 Command List Searching ... 147
13.2.1 Fences ... 147
13.2.2 Search Modes ... 147
13.2.3 Effective Search Mode ... 148
13.3 Commands Related to the Command List ... 148
13.3.1 change_command_search_mode (chacsm) ... 148
13.3.2 change_system_command_library (chascl) ... 149
13.3.3 create_command_list_entry(ies) (crecle) ... 149
13.3.4 delete_command_list_entry(ies) (delcle) ... 150
13.3.5 display_command_list (discl) ... 150
13.3.6 display_command_list_entry(ies) (discle) ... 150
13.4 Functions Related to the Command List ... 152
13.4.1 $command_list ... 152

13.4.2 $command_list_entry ... 152
13.4.3 $command_search_mode ... 153
13.4.4 $source ... 153
13.4.5 $source_of_caller ... 153

14 Command Utilities ... 154
14.1 UTILITY / UTILITYEND ... 154
14.1.1 Command ... 155
14.1.2 Function ... 156
14.1.3 TABLEND ... 156
14.2 Commands Related to Utilities ... 157
14.2.1 change_utility_attribute(s) (chaua) ... 157
14.3 Functions Related to Utilities ... 158
14.3.1 $utility ... 158
14.4 Interactive Include Processors (*) ... 158
14.5 Command Line Preprocessors (*) ... 159
14.6 An Example ... 159

15 format_scl_procedure (forsp) ... 162
15.1 Input to the Formatter ... 163
15.1.1 Pragmats ... 163
15.2 Utility Definition File ... 163
15.3 Output of the Formatter ... 165
15.4 An Example ... 166

16 Commands Related to Variables ... 168
16.1 create_default_variable (credv) ... 168
16.2 create_file_variable (crefv) ... 168
16.3 create_variable_list_entry(ies (crevle) (*) ... 169
16.4 delete_variable(s) (delv) ... 169
16.5 delete_variable_list_entry(ies) (delvle) (*) ... 169
16.6 display_variable_list (disvl) (*) ... 170
16.7 display_variable_list_entry(ies) (disvle) (*) ... 170

17 Create_Variable_Library(ies) (crevl) (*) ... 172
17.1 add_variable(s) (addv) (*) ... 172
17.2 combine_variable(s) (comv) (*) ... 172
17.3 replace_variable(s) (repv) (*) ... 173
17.4 add_library (addl) (*) ... 173
17.5 combine_library (coml) (*) ... 173
17.6 replace_library (repl) (*) ... 174
17.7 change_variable_description (chavd) (*) ... 174
17.8 delete_variable(s) (delv) (*) ... 175
17.9 display_library (disl) (*) ... 175
17.10 generate_library (genl) (*) ... 175
17.11 quit (qui) (*) ... 176

18 Miscellaneous Control Commands ... 177
18.1 change_interaction_style (chais) ... 177
18.2 change_message_level (chaml) ... 177
18.3 change_natural_language (chanl) ... 177
18.4 change_scl_option(s) (chasclo, chaso) ... 178
18.5 change_unseen_mail_action (chauma) ... 179
18.6 change_working_catalog (chawc) ... 179
18.7 collect_text (colt) ... 179
18.8 display_command_information (disci) ... 180
18.9 display_function_information (disfi) ... 182
18.10 display_command_stack (discs) (*) ... 183
18.11 display_value(s) (disv) ... 184
18.12 display_working_catalog (diswc) ... 184
18.13 get_line(s) (getl) ... 184

18.14 include_command (incc) 185
18.15 include_file (incf) 185
18.16 include_line (incl) 186
18.17 put_line(s) (putl) 187
18.18 wait 187

1 SCL Concepts

This document describes the NOS/VE Command Interface to be used by application programmers in either batch or interactive mode.

The System Command Language (SCL) is the fundamental means of communicating with NOS/VE; not just with the "system" itself, but also with the utility programs that run on the system. This means that when you begin to use a new utility on NOS/VE you don't have to learn a new language, only a new vocabulary—the commands provided by that utility.

SCL provides facilities for combining existing commands together to form new ones. In this sense it is a "programming" language as well as a "command" language. In addition to new commands, SCL can be extended by a number of other mechanisms such as functions, data types and command utilities.

The form of an SCL command is the same regardless of its origin: an interactive terminal, an SCL procedure or batch job. Use from a terminal is enhanced by various facilities such as prompting for omitted information, and the availability of "help" information for commands and parameters.

1.1 Statement Submission

SCL statements are submitted through interactive terminals, batch terminals and files. When an interactive terminal is used the statements are processed directly by the SCL interpreter, in which case, a slant "/" will be used as the basic prompt for input from the user. If a command line is continued the basic prompt is prefixed by two dots "../." These prompts are said to be basic prompts because when a command utility is active, the basic prompt is prefixed by a short sequence of characters that identify the utility.

When a batch terminal is used the statements are placed on mass storage for subsequent processing by the SCL interpreter. The interactive terminal or mass storage file is called the main command file for the job.

1.2 Statement Processing

A command line can contain one or more statements. Multiple statements on a single line must be separated by semicolons. It should be noted that the semicolon is a statement separator and not a statement terminator. A single statement placed on a line need not be followed by a semicolon.

1.3 Command Logging and Echoing

Commands read from the main command file for a job (file COMMAND) are written to the job log automatically by the SCL interpreter unless the command description inhibits it.

In addition to the job log, these commands are written to the system log provided that "system logging" has been activated (by default it is deactivated).

A command that has parameters that contain secure information such as passwords should inhibit the automatic logging of its call. A command that does this takes on the responsibility of producing an "edited" version of the call to it, then calling an SCL provided interface to log and/or echo the edited version. Although the responsibility for logging such a command rests with the command processor, if the processor uses the standard facilities for parameter evaluation, the logging and/or echoing is done as part of that process.

A command utility may choose to inhibit the logging of its own subcommands. Logging of other commands used within the utility still happens, however.

Echoing is the process of writing command images to a file as they are being processed to provide a trace of command procedure or command file interpretation. All commands that are executed from a file to which the user has read access can be echoed. Echoing is activated by creating a file connection from the standard file $ECHO to another file, and deactivated by deleting that file connection.

1.4 Commands

Commands are the most prevalent facility provided by SCL. The form of a command is a name optionally followed by one or more parameters. The parameters control what the command operates on and how it goes about doing its job. The packaging of the processor for a command (i.e. where it resides and the programming language in which it was implemented) is, in general, invisible and unimportant to the command user.

1.5 Functions

Functions are primarily used in the context of SCL as a programming language. They provide a convenient mechanism for obtaining information. The form of a function call is its name (which always begins with a dollar sign "$" character) optionally followed by one or more parameters (arguments) enclosed in parentheses. As with a command processor, the packaging of a function processor is, in general, invisible and unimportant to the caller of the function.

1.6 Control Statements

A number of control statements are provided to control the order of processing of commands. For example, the IF/ELSEIF/ELSE/IFEND statement is used to select one of a number of groups of statements to process; and the FOR/FOREND statement is used to repeat a group of statements.

1.7 Condition Handling and Status Parameters

Facilities are available to deal with unexpected situations. A status parameter can be supplied on most commands to receive the completion status of the command. This status parameter can then be interrogated by a control statement and appropriate action taken.

A more general mechanism is condition handling. In this scheme, when an unusual "condition" occurs, a "handler" for that condition is activated to deal with the condition. A condition handler is a sequence of statements, surrounded by the WHEN/WHENEND statement, that can determine why it was invoked and take appropriate action such as continue, abort or retry.

1.8 Parameters, Variables and Types

Parameters to commands and functions may be of many different types including file, name, number, string, etc. Also, these basic types may be grouped together into lists, ranges, arrays and records.

The value for any parameter of any command or function can be supplied either as a simple value or as an expression. The SCL interpreter evaluates and validates, on behalf of the command or function processor, that you have supplied appropriate values for the parameters; and, when being used interactively, prompts for omitted parameters or corrections when necessary.

Variables are often useful for control information, intermediate results, default values, aliases, etc. Variables can be of the same types as parameters to commands and functions.

Variables may be shared among jobs by storing them on "variable libraries." A variable library contains one or more SCL variables and may be shared according to the rules for sharing any file. One or more variable libraries may

be put into the "variable library list" which is treated, for purposes of accessing variables, as an extension to the main "block" (see below) in a job.

1.9 Files, Catalogs and Paths

A file is a named collection of data usually residing on a device such as a disk or tape. Each NOS/VE file has a set of attributes that describe its contents and usage. These attributes are usually established by the command or utility that creates the file and can be used to determine whether a file is being used in an appropriate manner or what the appropriate manner is.

NOS/VE provides a file system in which there are catalogs in addition to files. A catalog is a named collection of files and other catalogs. A catalog that is contained within another catalog is sometimes called a subcatalog and can itself contain files and still more catalogs. On some other systems, catalogs are known as directories.

This organization of files and catalogs is called a hierarchical file system. In such an organization, a simple name is not enough to uniquely identify a file or catalog. Rather a series of names is needed that starts at the top of the hierarchy, ends with the particular file or catalog of interest, and includes, in order, the names of all the subcatalogs between. Such a series of names is known as a path, or path name, since it describes how to "get to" the desired file or catalog.

The form of a path name in SCL is the list of names separated from each other by periods (dots, points). A path may be "absolute," that is completely spelled out, or "relative" to another catalog known as the "working catalog."

The "working catalog" concept provides a way for you to tell SCL: "This is the catalog where most of the files I'll need can be found." Then you don't have to give the complete path to those files, just their names within the working catalog.

Another use for command language variables is to act as aliases for path names. This is done by putting a path name in a variable and subsequently using the variable name whenever you need to reference the path name.

A number of functions are provided that designate frequently needed absolute path names.

1.10 Command Utilities

A command utility is a command that establishes an environment in which to perform a particular set of operations. Most important to this environment is a set of commands, known as the subcommands of the utility, through which the operations are actually performed. In addition command utilities often provide functions that ease the job of "programming" the utility.

One of the most attractive features of the environment created by a command utility is that all of the system or user supplied commands are still available while the utility is active. A command utility augments, rather than replaces, the environment that existed when the utility was called.

1.11 Command List

The command list is searched to find the processor for a command or function. An element of the command list can be an object library, a catalog, the subcommands and functions defined by a command utility, or the element, $SYSTEM, that designates the commands and functions supplied by the system. You may add to or delete from the command list to tailor your work environment.

1.12 Block Structure and Scope

SCL is a block structured programming language. Blocks, especially SCL procedures and utilities, provide the vehicle for controlling the scope of variables and condition handlers. The "scope" of an object refers to its residence (which block?), accessibility (from which blocks?), and lifetime.

1.13 Text Collection and Substitution

Certain statements, such as JOB/JOBEND, collect other statements for subsequent processing. The COLLECT_TEXT statement collects "arbitrary" text onto a file. Both situations occasionally require the substitution of parameter and variable values from the current environment. The facility to do this is provided via a substitution marker defined on the command.

1.14 Defaulting

The ease of use of commands and functions is enhanced by providing default values for most parameters. Defaults are defined in the parameter declarations for commands and functions and are evaluated as if they had been supplied on the command or function call.

Defaults for parameters can be established in variables referred to as "default variables." The default value for a parameter

that references a default variable can be changed simply by giving the default variable a new value.

1.15 Aliasing

On occasion the same path name, or other value or list of values must be used a number of times. In such situations an abbreviation or alias is a great convenience. To create such an alias simply put the desired value into a variable of the appropriate type.

1.16 Prologs and Epilogs

You can automatically have your environment customized when you login to NOS/VE by creating a file of commands to be processed at that time. Such a set of commands is called a prolog. Similarly you can arrange to have a set of commands automatically processed when you logout. This set of commands is called an epilog. Some utilities provide prolog/epilog facilities for more convenient use of the utility.

1.17 Message Templates

A message template contains most of the text of a status message. It is used with a status value to format a status message. The condition code in the status value is used to locate the descriptive information about the condition; this includes its severity and the template of the message. The text field of the status value usually contains parametric information that is included in the resulting message as directed by information contained in the template.

Message templates are contained in message modules which reside on an object library in the command list.

All information in a message module is assumed to be written in the same natural language and the currently selected natural language for the job is used in combination with the condition code from the status value to search for the actual condition description. Thus there may be message templates written in a number of natural languages for the same status condition.

If no condition description can be found corresponding to the preferred natural language for the job, but there is a description for the condition in US_English, it is used. Otherwise a template that causes the information in the status value to be formatted “raw” is used.

Message modules are created using the CREATE_MESSAGE_MODULE facility within the CREATE_OBJECT_LIBRARY utility.

1.18 Message Levels

The message level for a job determines the amount of information that is included in a message corresponding to a status.

When a command being processed at the main command level of a batch job or from the main command file of an interactive job terminates abnormally, the message corresponding to the status is formatted according to the job's message level and written to the $RESPONSE file. This file is connected to at least the job log and, for an interactive job, file OUTPUT.

There are two message levels: brief and full. Full level causes the status severity and condition code to appear at the beginning of the formatted message. In addition, any files or catalogs appearing in the message are always represented in “absolute path” format.

Brief level causes the condition code to be suppressed from the message. The severity is suppressed if it is informative. Also, any file or catalog that is within the working catalog is represented in “relative path” format.

1.19 Interactive Usage

Within an interactive job the SCL interpreter will, under appropriate circumstances, carry on a “dialog” with the user to obtain parameter values for a command. Such a “dialog” may take place in either “line mode” or “screen mode” depending on the circumstances that caused it to start, the job's “interaction style” and certain “SCL options.”

A job's interaction style, LINE or SCREEN, is selected via the CHANGE_INTERACTION_STYLE command. The default interaction style is LINE. The selected interaction style affects the way some applications and the SCL interpreter interact with the user.

The CHANGE_SCL_OPTIONS command provides two parameters for controlling the nature of the “parameter dialog” that occurs when a command's parameters are entered incorrectly. These options are described below.

The circumstances that can cause the SCL interpreter to begin a “parameter dialog” are:

1. A command is entered with parameters.

   This method of command entry starts a parameter dialog if a parameter is in error or if a required parameter is omitted. The mode of the dialog depends on the mode the terminal is in when the command is entered.

   If the terminal is in LINE mode, the SCL option LINE_STYLE_CORRECTION_PROMPTS determines the dialog's mode. The choices are NONE, LINE and SCREEN. The default is LINE.

   If the terminal is in SCREEN mode, the SCL option SCREEN_STYLE_CORRECTION_PROMPTS determines the dialog's mode. The choices are NONE and SCREEN. The default is SCREEN.

   NONE means that no parameter dialog occurs. An error is reported and the command must be re-entered. In a LINE mode dialog, errors are reported and corrections are prompted for until all required parameters have been supplied and all supplied parameters are satisfactory. The dialog and command can be cancelled using an interactive “terminate break.” SCREEN mode dialogs are described below.

2. A command is entered with no parameters but the command has one or more required parameters.

   This mode of the dialog is determined in the same way as for case 1, above. If the dialog is in LINE mode, the SCL interpreter prompts for values for *all* of the command's parameters. In this case the dialog is *always in* LINE mode.

   The dialog is similar to that described for case 1, above, except that instead of prompting just for required parameters, all parameters are prompted for. When an optional parameter is prompted for, pressing just the Next/Return key indicates the default value, if any, should be used for the parameter.

3. A “command with prompting statement” is entered. The form of this statement is a question mark (?) followed by a command reference.

   > NOTE that this statement may be used from within an SCL procedure or other non-interactive source of commands provided the job is interactive.

   This statement provides a way of forcing a dialog to occur. The dialog's mode is the currently selected interaction style for the job.

Messages and prompts used in both LINE and SCREEN mode are obtained from a “help module” associated with the command. If no help module can be located, the prompting information is derived from the data in the command's parameter description table (PDT).

If a function is entered for a parameter during a parameter dialog, and a mistake is made (such as described in cases 1 or 2, above), another parameter dialog in the same mode occurs to obtain corrections for the function's parameters. However, there is no analog to the “command with prompting statement” for functions.

1.19.1 Line Mode Dialogs

LINE mode does not require any special type of terminal but does take into account certain terminal attributes such as page width.

When entering corrections or omitted parameters the user may enter more than the parameter being prompted for. This is done in exactly the same way as parameters are normally given to a command.

In response to any prompt, an interactive “terminate break” can be entered to cancel processing of the command. Ending a response with a semicolon (;), or a response that consists entirely of a semicolon, indicates that no more prompting should occur and the command should be executed.

A question mark (?) can be entered to obtain information about acceptable responses to the prompt. Continuing to enter a question mark in response to a prompt will result in the display of various levels of available “help” information.

If the first character of a response is a slant (/) the rest of the response is interpreted as a command line.

1.19.2 Screen Mode Dialogs

SCREEN mode must be explicitly selected by the user via the CHANGE_INTERACTION_STYLE command and calling a command via the “command with prompting statement” (see above). SCREEN mode requires that the terminal definition for the type of terminal being used exists on a library in the job's library list. It is also necessary to have set the terminal_model terminal attribute that identifies the terminal definition.

As described above, a form is presented in which the user can “fill in” parameter values. Moving from one value entry area to the next is accomplished by normal cursor positioning, by the next/return key, or the tab key (for those terminals that support a “protected fields” feature).

The “OK” function key is used to indicate that parameter entry is complete and that command execution should begin.

The “CANCEL” function key is used to end the parameter dialog and execution of the command.

There are also function keys for requesting help, restoring a parameter's default value, etc.

1.19.3 “Desktop Environment” Interaction

A third interaction style is that available via microcomputers running CONNECT/VE's “Desktop Environment.” A description of this environment can be found in the (internal) document “NOS/VE Desktop

Environment User Interface, Preliminary Design Concept.” Briefly, it is (quoting from that document) “... a direct-manipulation interface to the applications available on [NOS/VE]. It replaces typed-in commands with a graphic-oriented interface that is much easier to learn and faster to use.”

This style of interaction is only available through the “desktop environment.”

1.19.4 Help

Information about available commands and functions may be obtained via the DISPLAY_COMMAND_LIST_ENTRY command.

Information about any command or function may be obtained via the DISPLAY_COMMAND_INFORMATION and DISPLAY_FUNCTION_INFORMATION commands.

1.20 Help Modules

A help module is a repository for prompts and messages associated with a command or function. A help module resides on an object library in the command list and is designated for use by a reference to it in the parameter description table (PDT) of the command or function.

All information in a help module is assumed to be written in the same natural language and the currently selected natural language for the job is used in combination with the help module reference in the PDT to search for the actual help module. Thus there may be help modules written in a number of natural languages for the same command or function.

If no help module can be found corresponding to the preferred natural language for the job, but there is a help module for the command or function in US_English, it is used. Otherwise prompts are derived from the information contained in the PDT for the command or function in conjunction with the prompt templates that can be found in the help module containing prompts and other messages issued by the system. This module's name is of the form CLM$SYS_MESSAGES$xxx where “xxx” is a natural language name.

Help modules are created using the CREATE_MESSAGE_MODULE facility within the CREATE_OBJECT_LIBRARY utility.

1.21 “Advanced” Commands, Functions, Parameters and Keywords

When you write a command or function you can designate a parameter as “advanced.” This has the effect of suppressing presentation of information about the parameter by the various information displays and help facilities, unless that information is explicitly requested. This same concept can be applied to commands and functions on an object library, and those belonging

to a utility.

1.22 “Hidden” Commands, Functions, Parameters and Keywords

When you write a command or function you can designate a parameter as “hidden.” This has the effect of suppressing presentation of information about the parameter by the various information displays and help facilities. This same concept can be applied to commands and functions on an object library, and those belonging to a utility.

This capability can be used to provide facilities for maintainers or “in house” users of commands that should not be made available to the general user community.

1.23 “Wild Card” Characters

Groups of files or names that adhere to a “naming convention” can be conveniently referenced using “wild card” characters. These characters specify a pattern that a name must match to be considered a member of the group.

The wild card characters can be used in any file or name expression that is part of a list expression. The wild card notation acts as a shorthand for specifying a list of files or names.

1.24 “Hooks” for Extension

SCL provides many ways to extend the language. The most common of these is the writing of commands and functions. In addition new data types can be defined in terms of existing ones or, for more unusual cases, “application values” can be defined.

2 Fundamentals of SCL

This section contains a complete specification of the System Command Language (SCL).

2.1 Lines and Their Continuation

From the standpoint of command interpretation there are two kinds of lines: command lines, which contain commands or control statements, and data lines, which can contain any text.

The distinction between the two is determined by the context in which they appear. For example, the collect_text command appears in a command line and reads the lines that follow the command line as data lines. The line following the last data line is another command line.

For any of a number of reasons it is sometimes necessary to break up a command line into two or more physical lines. You accomplish this by placing an ellipsis (two or more periods) at the end of all the physical lines, except the last, which comprise the broken up command line.

The construction of the command line from its component physical lines is a function of the SCL interpreter's input processing, not its language analysis. In other words, all of the component physical lines are read prior to any examination of the contents of the command line taking place.

```
Example: 'This is a long string constant th..
         at is continued from one line onto another.'
```

The length of a command line, after processing any continuation, is restricted to 65535 characters The length of a data line is also restricted to 65535 characters.

2.2 MetaLanguage

Throughout this language definition, SCL statements are illustrated with a uniform system of notation. This notation is called a metalanguage and is not part of SCL. The use of a metalanguage provides a brief explanation of the general patterns that SCL permits. The metalanguage does not describe the meaning of the language elements, merely their structure; i.e. it indicates the order in which the elements may (or must) appear, punctuation that is required, and options that are allowed.

Metalanguage Rules

1. The symbol ::= is read as “is defined to be.”
2. The symbol | is read as “or” and elements separated by it are mutually exclusive.
3. Elements enclosed by < > constitute a single element in relation to surrounding metalanguage symbols.
4. Elements enclosed by [ ] are optional and constitute a single element in relation to surrounding metalanguage symbols.
5. Elements followed by ... may be repeated. When the ... follows a > or], the ... applies to the metalanguage text between and including the matching < or [.
6. Underlined characters are to be interpreted literally in a metalanguage definition (e.g. “≤” means the character “<,” not the metasymbol “<”). An underline character by itself (e.g. _) means itself, as opposed to meaning the space character.
7. Names which appear in a metalanguage definition in upper case letters have, in the context in which they appear, special meaning to SCL. When using such names in actual SCL text, lower case letters may be used interchangably with the corresponding upper case letters.
8. The notation <the ascii xxxxx> is used to denote a particular non-graphic character in the ASCII (American Standard Code for Information Interchange) character set; xxxxx describes the character in question.
9. The notation <any ascii character except xxxxx> is used to denote any ASCII character except the character(s) specified by xxxxx.
10. The notation <xxxxx expression> is used to denote a expression of type xxxxx.

11. The notation <xxxxx function> is used to denote a function that returns a value of type xxxxx.
12. The notation <xxxxx command> is used to denote the command named xxxxx.
13. The notation <xxxxx name> is used to denote a name that designates a xxxxx.
14. The notation <xxxxx variable> is used to denote a variable of type xxxxx.
15. The notation <ascii> is used to designate any ASCII character.
16. The notation <beginning of line> is used to designate the beginning of a command line.
17. The notation <end of line> is used to designate the end of a command line.
18. The notation <eoi> is used to designate the end of information on a file.
19. The notation <eop> is used to designate the end of a partition on a file.
20. The notation <empty> is used to designate an empty (null) syntactic construct.

2.3 Building Blocks of SCL

The following sections define the building blocks out of which SCL text is constructed. These are variously referred to as language elements, syntactic units, lexical units, tokens, etc.

2.3.1 Comments

Comments may be used to document SCL text and are treated exactly like spaces. Comments begin with a double quote mark (") and may contain any character except a double quote mark. Comments not terminated with a double quote mark are terminated at the end of the command line.

```
<comment> ::= " [<any ascii character except ">]... ["]

Example: " This is a comment. "
         " This one is terminated by the end of the line.
```

The way in which line continuation is processed (see the section on “Lines and Their Continuation,” above) has an effect on how you must put together blocks of comments. We recommend that you not use continuation for lines within a block of comments in order to avoid the pitfall shown below.

```
         "This is an example of ..
         "a short block of ..
         "comments.
```

As a result of continuation processing, the above produces the following command line instead of the intended result.

```
         "This is an example of "a short block of "comments.
```

The intended result can easily be had by simply omitting the continuation ellipses from the first two lines in the example.

2.3.2 Spaces

Spaces may in general be used to separate other SCL elements from each other in order to improve readability. Also, SCL permits spaces to be used as separators between parameters and between the elements of a list. Whereever spaces may appear, comments may also be used.

```
<sp> ::= <spaces | comment>...

<spaces | comment> ::= <spaces> | <comment>

<spaces> ::= <space>...
```

```
<space> ::= <the ascii space character SP>
          | <the ascii horizontal tab character HT>
```

Because of their use as separators, it is necessary to impose restrictions on where spaces can occur. For instance, spaces are not permitted between the component parts of a file or variable reference.

How spaces are processed by the SCL interpreter is illustrated in the definitions in the following sections for other basic command language elements.

2.3.3 Delimiters

The following definitions are for delimiters used throughout SCL for various purposes. The definitions illustrate whether spaces preceding or following the actual delimiter are considered to be a part of the delimiter.

Delimiters other than those below are used in SCL for various purposes but only for those shown below is the use of surrounding spaces consistent throughout.

```
<,|sp|nl> ::= <,> [<nl>] | <sp> | <nl>

<,|;|nl> := <,> [<nl>] | <;> [<nl>] | <nl>

<sp|;|nl> := <sp> | <;> [<nl>] | <nl>

<sp|nl> ::= <sp> | <nl>

<;|nl> ::= <;> [<nl>] | <nl>

<nl> ::= <eol> <bol> [<eol> <bol>]...

<bol> ::= <beginning of line> [<sp>]

<eol> ::= [<sp>] <end of line>

<,|sp> ::= <,> | <sp>

<,> ::= [<sp>] , [<sp>]

<;> ::= [<sp>] ; [<sp>]

<:> ::= :

<(> ::= ( [<sp>]

<)> ::= [<sp>] )

<.> ::= .

<..> ::= [<sp>] <.><.> [<.>]... [<sp>]
```

2.3.4 Names

A name is a sequence of from 1 to 31 alphanumeric characters the first of which must not be a digit and which must be delimited at both ends. In a name the case of letters is irrelevant and all lower case letters appearing in a name are “folded” to their upper case counterparts.

```
<name> ::= <alphabetic char> [<alphanumeric char>]...

<alphanumeric char> ::= <alphabetic char> | <digit>

<alphabetic char> ::= <letter>
```

```
                    | <international letter>
                    | <special alphabetic char>

<letter> ::= <upper case letter>
           | <lower case letter>

<upper case letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M
                      | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<lower case letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m
                      | n | o | p | q | r | s | t | u | v | w | x | y | z

<international letter> ::= <upper case international letter>
                         | <lower case international letter>

<upper case international letter> ::= @ | [ | \ | ]| ^

<lower case international letter> ::= ` | { | | | } | ~

<special alphabetic_char> ::= # | $ | _

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Example: x, $dAtE, this_is_a_semi_long_name
```

2.3.4.1 Use of the $ in Names

As a general rule, you should not create a name that contains the dollar sign ($) character. This guideline is suggested because names containing the $ character have special significance to NOS/VE that depends on the context in which they are used. For example: function names, “keywords” in file references, system defined variables, and system created file names all contain a $ character.

2.3.4.2 Use of “International” Characters in Names

The so-called international characters may be used in NOS/VE names. These characters accomodate a general naming capability in those countries in which they are considered to be accented letters of the alphabet.

Users in countries within which the international characters are not considered to be alphabetic (such as the United States) are strongly urged to avoid employing these characters in the names they invent.

2.3.5 Integers

An integer constant is a sequence of digits, the first of which must be a decimal digit, optionally prefixed by a sign and optionally suffixed by a radix enclosed in parentheses. The constant must be delimited at both ends. No spaces are permitted between the digits and the radix specification.

```
<integer> ::= [<sign>] <unsigned integer>

<unsigned integer> ::= <digit> [<hex digit>]... [<(> <radix> <)>]

<sign> ::= <+|-> [<sp>]

<+|-> ::= + | -

<hex digit> ::= <digit> | A | B | C | D | E | F
                       | a | b | c | d | e | f

<radix> ::= <unsigned decimal>

<unsigned decimal> ::= <digit>...
```

An integer constant can be expressed in any radix between two and sixteen. When the radix specification is omitted, ten (decimal) is normally assumed. The default radix can be overridden when an integer type is defined so that whenever an integer literal is evaluated under control of such a type definition, the selected radix is assumed instead of the usually assumed decimal. (See the section on type definitions for more information.) Besides ten (decimal), the most common radix values are sixteen (hexadecimal), eight (octal) and two (binary). No distinction is made by the SCL interpreter between lower and upper case letters used to represent digits.

When a radix greater than ten is specified, a leading zero digit may be required to ensure that the constant begins with a decimal digit. This is to avoid ambiguity between, for example, the sixteenth element of the array whose name is A1 given by A1(16), and the hexadecimal representation of the decimal value 161 denoted by 0A1(16).

Spaces may not appear between the sign and unsigned integer in certain contexts, most notably in expressions for parameter values. See the syntax definition for numeric expressions for details.

```
Example: 63, 77(8), -63( 10), 3f(16), 111111(2), 0abc( 16 )
```

2.3.6 Real Numbers

A real number constant is a sequence of decimal digits containing a decimal point (period), an optional leading sign and an optional trailing exponent specification. The exponent is a sequence of decimal digits optionally prefixed by a sign and separated from the rest of the real number by the letter E or D in either upper or lower case. The constant must be delimited at both ends. No spaces are allowed between the mantissa and exponent specification, or within the exponent specification.

```
<real> ::= [<sign>] <unsigned real>

<unsigned real> ::= <mantissa> [<exponent>]

<mantissa> ::= <integer part> <.> <fraction part>

<integer part> ::= <unsigned decimal>

<fraction part> ::= <unsigned decimal>

<exponent> ::= <exponent char> [<+|->] <unsigned decimal>

<exponent char> ::= E | e | D | d
```

The real number value is formed by multiplying the mantissa by 10 raised to the power specified by the exponent. If the exponent is omitted none is used.

There is no significance given by the SCL interpreter to the particular exponent character (E or D, upper or lower case) used. SCL maintains real numbers in the normalized double precision floating point format.

Spaces may not appear between the sign and unsigned real in certain contexts, most notably in expressions for parameter values. See the syntax definition for numeric expressions for details.

```
Example: 123.456, -1.23456E2, 0.123456e+3, 123456.0d-3
```

2.3.7 Strings

A string constant is any (possibly empty) sequence of ASCII characters enclosed in apostrophes (single quote marks). The apostrophes are not part of the string but serve as delimiters. To include an apostrophe in the string, two consecutive apostrophes are used. The string must be delimited at both ends.

```
<string> ::= ' [<string char>]... '
<string char> ::= <any ascii character except '> | ''

Example: 'This is a string.'
         'Here is an enclosed apostrophe '' character.'
```

```
''        "a null (empty) string"
''''      "a string containing only an apostrophe"
```

2.3.8 Booleans

A boolean constant is represented by one of the names shown below. These names are interpreted as boolean values only in those contexts which require boolean values.

```
<boolean> ::= <true> | <false>

<true> ::= TRUE | YES | ON

<false> ::= FALSE | NO | OFF
```

3 Input to the SCL Interpreter

The following definitions illustrate the general form of the input processed by the SCL interpreter.

```
<SCL input> ::= <job>
              | <command procedure>
              | <function procedure>
              | <included file>
              | <included line>

<job> ::= <login> [<statement list>] <logout>

<login> ::= <batch login>
          | <interactive login>

<batch login> ::= <bol> [<;|nl>]... <login command> <;|nl>

<logout> ::= LOGOUT | <end of text>

<end of text> ::= <eol> <eoi> | <eol> <eop>

<included file> ::= <bol> [<statements>] <end of text>

<included line> ::= <bol> [<statements>] <eol>

<statement list> ::= <statements> <;|nl>

<statements> ::= <statement> [<;|nl> <statement>]...

<statement> ::= <command>
              | <command with prompting statement>
              | <type definitions>
              | <variable definitions>
              | <variable control statement>
              | <parameter statement>
              | <condition statement>
              | <structured statement>
              | <if statement>
              | <case statement>
              | <control statement>
              | <empty>

<variable control statement> ::= <assignment>
                               | <push>
                               | <pop>
                               | <lock>
                               | <unlock>

<parameter statement> ::= <check statement>

<condition statement> ::= <when statement>
                        | <cancel>
                        | <cause>
                        | <continue>

<structured statement> ::= <block statement>
                         | <repetitive statement>

<repetitive statement> ::= <for statement>
                         | <loop statement>
                         | <repeat statement>
                         | <while statement>

<control statement> ::= <cycle>
```

```
                    | <exit>
```

The various control statements are described in sections and subsections that follow.

4 Commands and Their Parameters

The following defintions illustrate the form of calls to commands.

```
<command> ::= <command reference> [<,|sp> <command parameters>]

<command reference> ::= <command name>
                      | / [<sp>] <command name>
                      | $SYSTEM <.> <command name>
                      | <utility name> <.> <command name>
                      | <file> <.> <command name>
                      | <catalog> <.> <command name> [<.> <cycle reference>]

<command parameters> ::= [<command parameter>] [<,|sp> [<command parameter>]]...

<command parameter> ::= [<parameter name> [<sp>] = [<sp>]] <parameter value>

<parameter value> ::= <expression>
                    | <omitted parameter name>
```

If only the <command name> is specified, the command list is searched for the command. See the section on the Command List for information on what constitutes the command list and how it is searched.

If the <command name> is prefixed by a slant character (/), the entries which precede a “fence” in the command list are bypassed in the search for the command. This facility is normally used within a command utility to call a command that doesn't belong to the utility but has the same name, usually abbreviation, as a utility subcommand.

If $SYSTEM is specified, the <command name> must designate one of the commands supplied as part of NOS/VE.

If a <utility name> is specified, the <command name> must designate a command established by the named utility. This form of command entry can only be used when the designated utility is currently active.

If a <file> is specified, the file must be an object library containing the command. A cycle reference, as part of the <file> specification, may be used to select a particular version of the object library. See the section on the Command List for information about how an object library is searched for a command.

If a <catalog> is specified, the <command name> must be the name of a file within the designated catalog that can be treated as a command. A cycle reference may be used to select a particular version of such a file. See the section on the Command List for information about what kinds of files can be regarded as commands.

Parameters for a command are specified as a sequence of individual parameters separated by commas or spaces. Parameters may be specified positionally or by name. When a parameter is not specified by name its position is taken to be one beyond that of the preceeding parameter. The significance of explicitly omitting a parameter (e.g. by placing two commas together) is only to establish the position of the next parameter. Giving a parameter by name has the effect of “tabbing” to that parameter position.

4.1 Use of Spaces in Parameter Values

Because spaces can be used to separate parameters, the use of spaces within an expression for a parameter value is somewhat constrained. Specifically, spaces may only be used within a string constant or within a parenthesized subexpression or to separate elements of a list.

4.2 Passing Omitted Parameters

Within an SCL procedure it is often necessary to pass parameters to the procedure on to commands or functions the procedure calls, or use parameters as initial values for variables defined within the procedure. This may be done with parameters omitted from the call to the outer procedure provided that the entire parameter is specified, i.e. the parameter is not qualified by a subscript, field reference, or substring reference. When this is done, from the point of view of the inner command or function, the corresponding parameter appears to have been omitted, and a variable appears to be uninitialized.

4.3 Command With Prompting Statement

The <command with prompting statement> provides a way of executing a command and having the command prompt for each of its parameters. The prompting will occur in line or screen mode depending on the interaction style selected for the job. This statement may only be used within an interactive job.

```
<command with prompting statement> ::= ?  [<sp>] <command reference>
```

5 Data Types

SCL supports a variety of data types and mechanisms for combining them into new data types.

5.1 Type Definitions

Types are established as illustrated by the type definition statement.

```
<type definitions> ::= TYPE <sp|;|nl>
                         <type definition> <;|nl>
                         [<type definition> <;|nl>]...
                       TYPEND

<type definition> ::= <type name> [<sp|nl>] <:|=> [<sp|nl>] <type expression>

<:|=> ::= <:> | =

<type expression> ::= <type function>
                    | <type variable>
                    | <type>

<type> ::= <elemental type>
         | <structured type>
         | <union type>
         | <application type>
         | <type specification type>
```

The restrictions imposed on the naming of variables also apply to the names given to types. See the section on Names of Variables for details.

NOTE that <type definitions> is in fact a specialized form of <variable definitions> (see the section on “Defining Variables” for complete information on <variable definitions>. For example the following type definition:

```
TYPE
  ascii_ordinal = integer 0..255
TYPEND
```

is equvialent to the following variable definition.

```
VAR
  ascii_ordinal: (READ, ENVIRONMENT) type = integer 0..255
VAREND
```

5.2 Elemental Types

```
<elemental type> ::= <boolean type>
                   | <COBOL name type>
                   | <data name type>
                   | <file type>
                   | <integer type>
                   | <keyword type>
                   | <lock type> (*)
                   | <name type>
                   | <program name type>
                   | <real type>
                   | <statistic code type>
                   | <status code type>
```

```
                  | <string type>
                  | <string pattern type> (*)
```

5.2.1 Boolean Type

```
<boolean type> ::= BOOLEAN
```

5.2.2 COBOL Name Type

A “COBOL name” is a name whose syntax is acceptable in the programming language COBOL but does not conform to the SCL syntax for names. A COBOL name may contain the hyphen character and may start with a digit, both of which disqualify it from being an SCL name.

The general use of COBOL names is strongly discouraged. They are provided primarily for use in situations where COBOL generated names may need to be referenced, e.g. module and entry point names in the execute_task and related commands, and program identifiers within a DEBUG session.

```
<COBOL name type> ::= COBOL_NAME
```

5.2.3 Data Name Type

The data name type is used for designating a function name, parameter name or variable name without evaluating the function, parameter or variable.

```
<data name type> ::= DATA_NAME
```

5.2.4 File Type

The file type is used for files, with optional cycle reference and position designator, catalogs and paths that may designate either files or catalogs.

```
<file type> ::= FILE
```

5.2.5 Integer Type

```
<integer type> ::= INTEGER [<sp> <integer subrange>] [<default radix>]

<integer subrange> ::= <min integer> <..> <max integer>

<min integer> ::= <integer expression>

<max integer> ::= <integer expression>

<default radix> ::= <sp> RADIX <sp> <radix>
```

The subrange specification can be used to restrict the range of values applicable for the type being defined. The value of <min integer> must be less than or equal to that of <max integer>.

The radix specification can be used to select the default radix for integer literals that are evaluated under control of the type being defined. Omission of the radix specification causes 10 (decimal) to be assumed.

5.2.6 Keyword Type

The keyword type is generally used for designating a set of options.

```
<keyword type> ::= KEY <sp|nl>
                     <keyword groups>
                   [ADVANCED_KEY <sp|nl>
                     <keyword groups>]
                   [HIDDEN_KEY <sp|nl>
                     <keyword groups>]
                   KEYEND

<keyword groups> ::= <keyword group> <,|sp|nl> [<keyword group> <,|sp|nl>]...

<keyword group> ::= <(> <keyword> [<,|sp|nl> <keyword>]... <)> | <keyword>

<keyword> ::= <name>
```

The secondary keywords of a keyword group are assumed to be aliases for the first keyword of the group. The result of an expression for a keyword type is the name of the first keyword of the corresponding group.

5.2.6.1 Advanced Keywords

Keywords defined following the optional ADVANCED_KEY specifier in a keyword type are not shown in information displays unless explicitly requested.

5.2.6.2 Hidden Keywords

Keywords defined following the optional HIDDEN_KEY specifier in a keyword type are not advertised. For example, in a display of the allowed values for a keyword type parameter, all keywords with the “hidden” attribute are not shown.

5.2.7 Lock Type (*)

The lock type is used to synchronize the actions of asynchronous processes, be they jobs or tasks.

```
<lock type> ::= LOCK
```

5.2.8 Name Type

```
<name type> ::= NAME [<sp> <name size>]

<name size> ::= [<min name size> <..>] <max name size>

<min name size> ::= <integer expression>

<max name size> ::= <integer expression>
```

The name size specification can be used to restrict the length of names applicable for the type being defined. The value of <min name size> must be less than or equal to that of <max name size> and both must be in the range 1..$max_name. If <min name size> is omitted, 1 is assumed. If <max name size> is omitted, $max_name is assumed.

5.2.9 Program Name Type

A program name is used to designate an entry point or module name of an executable program.

Program_name is a “built-in” type whose definition is as follows (see the section on union types).

```
TYPE
  program_name: ANY OF
    NAME
    COBOL_NAME
    STRING 1..$max_name
  ANYEND
TYPEND
```

Hence the following definition.

```
<program name type> ::= PROGRAM_NAME
```

The string form of a program name can be used to specify an entry point or module name whose syntax does not conform to that of a standard SCL name or a COBOL name. In the string form the case of letters is significant, i.e. folding of lower case letters to uppercase does not take place as it does with SCL and COBOL names.

5.2.10 Real Type

```
<real type> ::= REAL [<sp> <real subrange>]

<real subrange> ::= <min real> <..> <max real>

<min real> ::= <real expression>

<max real> ::= <real expression>
```

The real type is represented as a normalized double precision floating point number.

The subrange specification can be used to restrict the range of values applicable for the type being defined. The value of <min real> must be less than or equal to that of <max real>.

5.2.11 Statistic Code Type

```
<statistic code type> ::= STATISTIC_CODE
```

The statistic code type is used to specifiy a code for a statistic. Such codes are stored internally as integers but are thought of as consisting of two parts: a two character identifier and a number. This type provides a mechanism for specifying a statistic code in a number of convenient formats.

5.2.12 Status Code Type

```
<status code type> ::= STATUS_CODE
```

The status code type is used to specifiy a condition code for a status. Such codes are stored internally as integers but are thought of as consisting of two parts: a two character identifier and a number. This type provides a mechanism for specifying status condition code in a number of convenient formats.

5.2.13 String Type

```
<string type> ::= STRING [<sp> LITERAL] [<sp> <string size>]

<string size> ::= <min string size> <..> <max string size>
                | <fixed string size>
```

```
<min string size> ::= <integer expression>

<max string size> ::= <integer expression>

<fixed string size> ::= <integer expression>
```

The string size specification can be used to restrict the length of strings applicable for the type being defined. The value of <min string size> must be less than or equal to that of <max string size> and both must be in the range 0..$max_string. If the string must be of one particular length, the <fixed string size> specification can be used to give values for both <min string size> and <max string size>. If the <string size> specification is omitted, 0 is assumed for <min string size> and $max_string for <max string size>.

The LITERAL qualifier is used in situations where it is desirable not to allow the general string type. (An example is the UNTIL parameter of the COLLECT_TEXT command.) It constrains the “expression” for the type to consist solely of a <string>.

The guideline for deciding whether to include the LITERAL qualifier is: don't.

5.2.14 String Pattern Type

String patterns are used for string search operations, for example in text editors.

```
<string pattern type> ::= STRING_PATTERN
```

5.3 Structured Types

```
<structured type> ::= <array type>
                    | <command reference type>
                    | <date time type>
                    | <entry point reference type>
                    | <line identifier type>
                    | <list type>
                    | <range type>
                    | <record type>
                    | <status type>
                    | <time increment type>
                    | <time zone type>
```

5.3.1 Array Type

Arrays should be used for structuring data whose elements are to be accessed randomly. The selection of an element is by an array index value, called a subscript. The value of the array's lower_bound must be less than or equal to that of its upper_bound. A subscript must be in the range lower_bound to upper_bound. The size of the array is upper_bound - lower_bound + 1.

```
<array type> ::= ARRAY [<sp> <array bounds>] [<array type qualifier>]

<array bounds> ::= <lower_bound> <..> <upper_bound>

<array type qualifier> ::= <sp> OF <sp> <type expression>

<lower_bound> ::= <integer expression>

<upper_bound> ::= <integer expression>
```

The array bounds may only be omitted when defining a type for a parameter of a command or function. The omission

means that the parameter may be passed an array with any bounds.

The array type qualifier may only be omitted when defining a type for a parameter of a command or function. The omission means that the parameter may be passed an array with any element type.

5.3.2 Command Reference Type

A command reference is used to designate an SCL command or statement name.

Command_reference is a “built-in” record type whose definition is as follows (see the section on record types).

```
TYPE
  command_reference: RECORD
    name: NAME
    form: KEY
      name_only
      skip_first_entry
      system
      utility
      module_or_file
      file_cycle
    KEYEND
    utility: NAME
    library_or_catalog: FILE
    cycle_number: INTEGER
  RECEND
TYPEND
```

Hence the following definition.

```
<command reference type> ::= COMMAND_REFERENCE
```

The keyword values of the form field correspond to the various forms of a <command reference> as shown in the following table. The table also shows which of the fields other than name and form are meaningful for the various forms.

Keyword	Form	Other Fields
name_only	<command name>	
skip_first_entry	/ [<sp>] <command name>	
system	$SYSTEM <.> <command name>	
utility	<utility name> <.> <command name>	utility
module_or_file	<file> <.> <command name>	library_or_catalog
	| <catalog> <.> <command name>	
file_cycle	<catalog> <.> <command name> <.> <cycle reference>	library_or_catalog cycle_number

5.3.3 Date Time Type

The date_time data type is used to designate a specific point in time. It is a “built-in” record type whose definition is as follows (see the section on record types).

```
TYPE
  date_time: RECORD
    year: INTEGER 1900..2155 = $OPTIONAL
    month: INTEGER 1..12 = $OPTIONAL
    day: INTEGER 1..31 = $OPTIONAL
    hour: INTEGER 0..23 = $OPTIONAL
    minute: INTEGER 0..59 = $OPTIONAL
```

```
            second: INTEGER 0..59 = $OPTIONAL
            millisecond: INTEGER 0..999 = $OPTIONAL
          RECEND
        TYPEND
```

Hence the following definitions.

```
<date time type> ::= <date time type identifier> [<sp> <date time type qualifier>]

<date time type identifier> ::= DATE_TIME | DATE | TIME

<date time type qualifier> ::= <tense> [<,|sp> <tense>]

<tense> ::= FUTURE | PAST | PRESENT
```

Specifying DATE_TIME means that both the date and time components of a date_time expression are significant. Specifying DATE means that only the date components of a date_time expression is significant, if a time component is given it is ignored and all time subcomponents are set to zero. Specifying TIME means that only the time components of a date_time expression is significant, if a date component is given it is ignored and the date is set to today's date.

One or more tenses may be used to qualify the type and are used to validate the result of a date_time expression. If no tenses are specified any legitimate date_time value is acceptible.

5.3.4 Entry Point Reference Type

An entry point reference is used to designate a particular entry point on an object library. A number of file attributes are entry point references; for example, the file access procedure (FAP) for a file is specified as an entry point reference.

Entry_point_reference is a “built-in” record type whose definition is as follows (see the section on record types).

```
      TYPE
        entry_point_reference: RECORD
          entry_point: NAME
          object_library: FILE = $OPTIONAL
        RECEND
      TYPEND
```

Hence the following definition.

```
<entry point reference type> ::= ENTRY_POINT_REFERENCE
```

5.3.5 Line Identifier Type

A line identifier is used to designate a line of text in a Source Code Utility (SCU) “deck.”

Line_identifier is a “built-in” record type whose definition is as follows (see the section on record types).

```
      TYPE
        line_identifier: RECORD


  modification_name: NAME 1..9
            sequence_number: INTEGER 1..0ffffff(16)
          RECEND
      TYPEND
```

Hence the following definition.

```
<line identifier type> ::= LINE_IDENTIFIER
```

See the SCU ERS, DCS# ARH3883, for more details on the meaning and usage of line identifiers.

5.3.6 List Type

Lists should be used for structuring data whose elements are, in general, to be accessed sequentially, or in cases where the actual number of elements in the list will vary from one usage to the another.

```
<list type> ::= LIST [<sp> REST] [<sp> DEFER_EXPANSION] [<list type qualifier>]

<list type qualifier> ::= [<sp> <list size qualifier>] <sp> OF <sp> <type expression>

<list size qualifier> ::= <min list size> <..> <max list size>

<min list size> ::= <integer expression>

<max list size> ::= <integer expression>
```

The list must have at least <min list size> elements. If <min list size> is not specified, one is assumed for the type of a command or function parameter, and zero otherwise. The list may not have more than <max list size> elements. If <max list size> is not specified, $max_list is assumed.

The list type qualifier may only be omitted when defining a type for a parameter of a command or function. The omission means that the parameter may be passed a list with any element type.

The REST qualifier may only be given when defining a type for the last field of a record, or the last parameter of a function. For a function, the parameter may not have the VAR attribute. If present it signifies that all remaining expressions are to be treated as expressions for the elements of a list which becomes the value of the field or function parameter. It can be thought of as: “make a LIST out of the REST of the field or parameter values supplied for the record or function.” (See the section on list expressions for more information.

The `DEFER_EXPANSION` option is used to prevent the expression evaluator from “expanding” an element expression containing “wild cards.” This option is only meaningful when the element type of the list is file, name, data_name, program_name or keyword. When such an “expression” for such an element is evaluated using this option, the wild card pattern itself becomes the value of the element and is produced as an application value.

5.3.7 Range Type

A range type is used to specify a pair of values that have some order relationship between them. For example, the <subrange> for an integer type is a range. The ordering relationship is imposed by the context in which the range is used rather than as a general property of the range structure.

```
<range type> ::= RANGE [<range type qualifier>]

<range type qualifier> ::= <sp> OF <sp> <type expression>
```

The range type qualifier may only be omitted when defining a type for a parameter of a command or function. The omission means that the parameter may be passed a range with any element type.

5.3.8 Record Type

Records provide a structuring mechanism for grouping data items of different types together. Each item is called a field of the record and is referenced via its field name.

```
<record type> ::= RECORD <sp|nl>
                     <field definition> <,|;|nl>
                    [<field definition> <,|;|nl>]...
                  RECEND
```

```
<field definition> ::= <field name> [<sp|nl>] <:> [<sp|nl>]
                       <type expression> [[<sp|nl>] = [<sp|nl>]
                       <field requirement>]

<field requirement> ::= $REQUIRED
                      | $OPTIONAL
```

The <field requirement> specifies whether a value of the record type is complete without the associated field. Completeness is checked for on assignment to a variable or parameter of the record type. $REQUIRED indicates that a value must be provided for the corresponding field for the record value to be considered complete. $OPTIONAL indicates that a value need not be specified for the field. If the <field requirement> is omitted, $REQUIRED is assumed.

5.3.9 Status Type

A status type is used to capture the completion status of a command.

Status is a “built-in” record type whose definition is as follows (see the section on record types).

```
TYPE
  status: RECORD
    normal: BOOLEAN
    condition: INTEGER 0..0fffffffff(16) = $OPTIONAL
    text: STRING 0..256 = $OPTIONAL
  RECEND
TYPEND
```

Hence the following definition.

```
<status type> ::= STATUS
```

The fields of the status record are used as follows:

NORMAL	A value of TRUE indicates that the command that generated the status completed successfully. In this case the remaining fields of the status record are meaningless. A value of FALSE usually indicates that the command could not be completed successfully. FALSE may also mean that the status represents some useful information concerning the command's successful completion.
CONDITION	This numeric code uniquely identifies the abnormal or informative status. This field is meaningless if the NORMAL field is TRUE.
TEXT	This 256 character string is used to hold information to be substituted into the message corresponding to the condition specified by the status record. The first character in this field is the character used to delimit the “message parameters” contained in the text field. Each sequence of characters that is preceded by the delimiter is considered a separate message parameter that is edited into the formatted message. Normally the delimiter character is the ASCII unit separator character: US, $char(31). This field is meaningless if the NORMAL field is TRUE.

5.3.10 Time Increment Type

The time increment type is used to designate an amount of time. It is a “built-in” record type whose definition is as follows (see the section on record types).

```
TYPE
  time_increment: RECORD
    years: INTEGER
    months: INTEGER
```

```
        days: INTEGER
        hours: INTEGER
        minutes: INTEGER
        seconds: INTEGER
        milliseconds: INTEGER
      RECEND
    TYPEND
```

Hence the following definition.

```
<time increment type> ::= TIME_INCREMENT
```

5.3.11 Time Zone Type

The time zone type is used to designate the difference between the time in some part of the world from “universal” time (Greenwich Mean Time). A time zone type can be thought of as a specialized version of the time increment type. It is a “built-in” record type whose definition is as follows (see the section on record types).

```
TYPE
  time_zone: RECORD
    hours_from_gmt: INTEGER -12..12
    minutes_offset: INTEGER -30..30
    daylight_saving_time: BOOLEAN
  RECEND
TYPEND
```

Hence the following definition.

```
<time zone type> ::= TIME_ZONE
```

The hours_from_gmt field specifies the number of hours west (negative) or east (positive) of Greenwich Mean Time (GMT).

The minutes_offset field is used to accomodate places such as Newfoundland, Java and southern Australia which are offset from a neighboring time zone by thirty minutes.

The daylight_saving_time field speciifes whether daylight saving time is in effect in the time zone (TRUE) or not (FALSE). The net affect of daylight_saving_time being TRUE is to add one to the hours_from_gmt field.

A date_time value does not carry with it any information concerning the time zone it is relative to. Each NOS/VE system uses a particular time zone to which any dates or times returned by the system are relative.

5.4 Union Type (Any)

The union type provides for the case where any one of a number of types is applicable.

```
<union type> ::= ANY [<union type qualifier>]

<union type qualifier> ::= <sp> OF <sp|nl>
                             <member definition> <,|;|nl>
                           [<member definition> <,|;|nl>]...
                         ANYEND

<member definition> ::= <type expression>
```

If the union type qualifier is omitted the union consists of all possible types.

The order in which the members of the union type are defined is significant. If an expression for a union type can be

successfully interpreted for more than one of its member types, it is given the first such interpretation.

```
Example: var
           x: any of
             name
             file
           anyend
         varend

         x = fred
         display_value $type(x)
         NAME
         x = $working_catalog.fred   "or $wc.fred"
         display_value $type(x)
         FILE
```

5.5 Application Type

Application types may be defined to deal with those situations where no other SCL supplied type will do.

```
<application type> ::= APPLICATION [<sp> BALANCE_BRACKETS]
```

It is the responsibility of a command or function processor that is passed an expression for an application value to parse and evaluate that expression.

If the BALANCE_BRACKETS attribute is specified, occurrences of the square bracket characters (“[” and “]”) and the brace bracket characters (“{” “}”) must be balanced. See the section on “Application Expressions.”

5.6 Type Specification Type

A type specification type is used to represent the result of a type expression. Type definitions are represented by environment variables whose type is type specification.

```
<type specification type> ::= TYPE
```

6 Variables

A variable is a named container of data. SCL supports two classes of variables known as procedure and environment.

Procedure variables are used for the normal programming language purposes such as holding results of computations, passing information between cooperating commands, etc. Such variables are not generally used directly by end users but appear in command and function procedures.

Environment variables are used to hold information about the environment in which a user is operating. Typical uses for environment variables are to hold the values of such things as user defined aliases for file or catalog names, user defined defaults for parameters, user defined data types, and the names of the object libraries referenced in object modules generated by the standard compilers.

Environment variables may be selectively stored on files called “variable libraries.” This capability is used, for example, to provide for sharing variables between jobs.

6.1 Defining Variables

Variables are established by the variable defintion statement.

```
<variable definitions> ::= VAR <sp|;|nl>
                               <variable definition> <;|nl>
                              [<variable definition> <;|nl>]...
                           VAREND

<variable definition> ::= <variable name expression> [<sp|nl>] <:> [<sp|nl>]
                          [<(> <variable attributes> <)> [<sp|nl>]]
                          <type expression> [[<sp|nl>] = [<sp|nl>]
                          <initial value>]

<variable name expression> ::= <data name expression>

<variable attributes> ::=
<variable attribute> [<,|sp> <variable attribute>]...

<variable attribute> ::= <variable scope>
                       | READ
                       | DEFER

<variable scope> ::= <procedure variable scope>
                   | <environment variable scope>

<procedure variable scope> ::= LOCAL
                             | XDCL
                             | XREF

<environment variable scope> ::= ENVIRONMENT
                               | JOB
                               | TASK
                               | UTILITY
                               | PUSH

<initial value> ::= [<initial name> <,> [<nl>]] <expression>
```

6.2 Names of Variables

Certain restrictions are imposed on the names variables can have.

1. The boolean constant names, TRUE, FALSE, YES, NO, ON and OFF, cannot be used for variable names.

2. A variable name cannot begin with the dollar sign ($) character.
3. A variable should not be given a name that is being used for a parameter in the same block. For compatibility with previous versions of NOS/VE, however, this is allowed.

6.3 Initial Values for Variables

Variables can be given an initial value at the time they are established. A variable must be given a value, either when it is defined or by an assignment statement or some other mechanism, before it can be read.

An <initial name> specifies the name of an optionally defined string variable. If at the time the variable is being declared the name is defined, its associated string is interpreted as the expression for the initial value. If the name is not defined the <initial value>'s expression is used. (This is analogous to the <default name> capability for command and function parameters.)

For compatibility with previous versions of NOS/VE, variables declared that are of certain types have default initial values. These are shown in the following table.

Type	Default Initial Value
BOOLEAN	FALSE
INTEGER	0
REAL	0.0
STATUS	NORMAL field = TRUE
STRING	" (null string)

In addition, variables that are ARRAYs OF any of the above types are, by default, initialized such that each array element is given the appropriate value from the above table.

A variable of type LIST with a minimum size of zero has a default initial value of an empty list

Variables of a type not mentioned above are not given initial values. Also, these default values are only used in those cases where they are allowed values for the variable. For example, for a variable whose type is “INTEGER 1..10,” the value 0 is not allowed, so that variable would have no default initial value.

It is good programming practice to explicitly specify an initial value for a variable, if appropriate. The intent of the programmer will be clearer thereby yielding a more maintainable result.

6.4 Read-Only Variables

Variables can be established as read-only so that they cannot be modified after their definition. A variable defined with the READ attribute must be given an initial value.

6.5 Residence of Variables

Variables can reside in the following kinds of SCL blocks.

JOB	This is the current block once a job has begun execution.
PROCEDURE	A procedure block is established for the execution of an SCL command or function procedure.
TASK	A task block exists for every task in a job. Variables may only reside in certain task blocks: those created for a task initiated by the TASK/TASKEND statement, and those created for any asynchronous task.
UTILITY	A utility block is established and named by a command utility.
WHEN/WHENEND	A when/whenend block is established at the time a condition occurs for which a when/whenend condition handler has been defined.

A procedure variable resides in the block in which its definition occurred.

An environment variable resides in the block designated by the environment variable scope specified when the variable

was defined. See Scope of Environment Variables, below, for details.

6.5.1 The Variable Library List (*)

In addition to residing within a job's environemnt, variables may be placed on “variable library” files and the file added to a job's “variable library list.” This is a list of files that are, in effect, treated as an extension to the JOB block (see above).

When searching for a variable in the JOB block, variables created within the job in that block are searched first. If the variable is not found, the files in the variable library list are searched in turn until either the variable is found, or the list is exhausted.

The commands CREATE_VARIABLE_LIST_ENTRY(IES) and DELETE_VARIABLE_LIST_ENTRY(IES) are used to manipulate the contents of the variable library list. The CREATE_VARIABLE_LIBRARY utility is used to create and/or change libraries of variables. These commands are described in the section on control commands.

6.6 Accessibility of Variables

A variable is always accessible within the block in which it resides. It may also be accessible within blocks called directly or indirectly by that block. Access to variables residing in a block other than the referencing block can occur implicitly or explicitly.

Implicit access exists to all environment variables residing in blocks that directly or indirectly called the referencing block.

Implicit access also exists to all procedure variables residing in blocks to which the referencing block is statically linked. A block is statically linked to another block if the former is established directly within the latter. For example, a when/whenend block is statically linked to the block that specifies the when/whenend sequence to handle a condition; and a utility block is statically linked to the block that calls the utility.

Explicit access is only necessary for procedure variables and is achieved by defining the variable in the referencing block with a scope of XREF. For this to work the variable in question must have been initially defined with the XDCL scope attribute.

6.7 Lifetime of Variables

A variable exists from the time it is defined until either it is explicitly deleted or popped, or until the block in which it resides terminates.

6.8 Scope of Procedure Variables

Some aspects of the scope of procedure variables have already been discussed in the sections on variable residence and accessibility. A summary of the procedure variable scope attributes follows.

LOCAL	This is the default scope attribute. A variable with this scope attribute is accessible only within the block in which it resides and blocks statically linked to it.
XDCL	A variable with this scope attribute is accessible within the block in which it resides, blocks statically linked to it, and blocks called directly or indirectly by it which define the variable with the XREF scope attribute.
XREF	This scope attribute is used to access a variable residing in a block which directly or indirectly called the referencing block. The variable must have been initially defined with the XDCL scope attribute. A variable defined with the XREF scope attribute may not be given an initial value. (For compatibility with previous system versions, the XREF scope may also be used to access a variable created with any of the “environment” scopes” discuused below.)

6.9 Scope of Environment Variables

To create an environment variable, an environment scope attribute must be specified in the variable definition. Specifying any of the environment scope attributes in a variable definition causes the variable to be defined as an environment variable. Each environment scope attribute designates the block in which the variable will reside as described below. The scope attributes of JOB, TASK and UTILITY can cause a variable to be created in a block other than the referencing block. In such cases two "accesses" to the variable are created, one in the block in which the variable resides, and one in the referencing block.

ENVIRONMENT	Specifiying this scope attribute causes the variable to reside in the defining block.
JOB	Specifying this scope attribute causes the variable to reside in the JOB block.
TASK	Specifying this scope attribute causes the variable to reside in the referencing block if it is a task block, otherwise in the task block that most directly called the referencing block.
UTILITY	Specifying this scope attribute causes the variable to reside in the referencing block if it is a utility block, otherwise in the utility block that most directly called the referencing block.
PUSH	This is, strictly speaking, not a true scope. It is provided as a convenience for defining environment variables in a context where a definition may or may not have already been made. If the variable has already been defined as an environment variable, it is PUSHed (see the PUSH statement for information on pushing an environemnt variable). The type must be identical to the original definition. If an initial value is specified, it becomes the value of the PUSHed instance of the variable. If the variable has not already been defined, it is defined with a scope of ENVIRONMENT.

6.9.1 Example of PUSH Scope

In the following partial procedure:

```
PROCEDURE example (
  ...
  feature_catalog, fc: file = wef$feature_catalog, $null
  ...
  )
  ...
  VAR
    wef$feature_catalog: (PUSH) string = ..
          $string(feature_catalog)
  VAREND
  ...
PROCEND example
```

the declaration of variable wef$feature_catalog is equivalent to:

```
IF $variable(wef$feature_catalog, environment) THEN
  PUSH wef$feature_catalog
  wef$feature_catalog = $string(feature_catalog)
ELSE
  VAR
    wef$feature_catalog: (ENVIRONMENT) string ..
          $string(feature_catalog)
  VAREND
IFEND
```

6.10 Deferred Evaluation of Variables

If a variable is defined with the DEFER attribute, the evaluation of any expression assigned to it is deferred until the variable is referenced, and reevaluated each time it is referenced. This, of course, requires that the expression assigned to such a variable be capable of being evaluated in every context in which the variable is to be referenced.

6.11 Variable References

The following definitions illustrate how variables are referenced.

```
<variable> ::= <variable name> [<value qualifier>]...

<value qualifier> ::= <parenthesized value qualifier>
                    | <field qualifier>
                    | <range qualifier>

<parenthesized value qualifier> ::= <subscript qualifier>
                                  | <substring qualifier>

<subscript qualifier> ::= <(> <subscript expression> <)>

<subscript expression> ::= <integer expression>

<field qualifier> ::= <.> <field name>

<range qualifier> ::= <.> <range selector>

<range selector> ::= LOW | HIGH

<substring qualifier> ::= <(> <substring index> <,|sp> <substring length> <)>

<substring index> ::= <integer expression>

<substring length> ::= <integer expression>
                     | ALL
```

A subscript qualifier is used to select a particular element of an array or list. The subscripts for a list are in the range 1..$size(the_list).

A field qualifier is used to select a particular field of a record structure.

A range qualifier is used to select one element of a range value. LOW selects the “low” value of the range and HIGH selects the “high” value. If the “high” value was not specified when the range value was generated, HIGH is equivalent to LOW.

A substring qualifier is used to select a string of characters from within string value. The substring selected consists of the <substring length> characters beginning with the character at position <substring index>. The ALL keyword designates that all characters from the substring index to the end of the string are to be selected.

6.12 Variable Control Statements

6.12.1 Assignment

The assignment statement is used to give a value to a variable or component of a variable. If the variable does not exist it is created as a LOCAL procedure variable whose type is determined by the expression being assigned. A variable that has the read attribute cannot be assigned to.

```
<assignment> ::= <variable> [<sp>] = [<sp>] <expression>
```

Unless the type of the variable is STRING the expression must conform to all of the specifications of the type. If a string variable is being assigned to, the expression may represent a string that is longer than the maximum size allowed by the type, in which case excess characters are truncated, or a string that is shorter than the minimum size allowed, in which case padding with space characters occurs.

6.12.2 PUSH

A new version of an existing environment variable can be created by the PUSH statement. The new version of the environment variable resides in the SCL block in which the push is requested and is given the same value as the previous version of the variable.

The pushed version of the variable does not have the READ attribute even if the previous version did. The pushed version does, however, inherit the DEFER attribute, or lack of it, from the previous version.

The PUSH statement can also be used to create new versions of certain “system environment” objects. These objects can be pushed and popped just like environment variables, but are otherwise manipulated by statements, commands and/or functions specific to them.

Only the most recently pushed version of an environment variable or system environment object can be referenced or changed.

```
<push> ::= PUSH <sp> <environment object> [<,|sp> <environment object>]...

<environment object> ::= <system environemnt object>
                       | <environment variable name>

<system environment object> ::= ATTACH_FILE_DEFAULTS (*)
                              | COMMAND_LIST
                              | FILE_ATTRIBUTE_DEFAULTS (*)
                              | FILE_CONNECTIONS
                              | INTERACTION_STYLE
                              | LINK_ATTRIBUTES
                              | MESSAGE_LEVEL
                              | NATURAL_LANGUAGE
                              | PROGRAM_ATTRIBUTES
                              | UNSEEN_MAIL_ACTION
                              | WORKING_CATALOG
```

6.12.3 POP

The POP statement deletes the current version of an environment variable or system environment object and restores the previous version. This statement cannot be used to delete the original definition of the variable. The pop operation is performed automatically upon exit from the block in which the corresponding push operation occurred.

```
<push> ::= POP <sp> <environment object> [<,|sp> <environment object>]...
```

6.12.4 LOCK (*)

The LOCK statement is used to set a lock variable.

```
<lock> ::= LOCK <sp> <lock variable> <sp> <lock qualifier>

<lock qualifier> ::= WAIT
                   | SET <sp> <boolean variable>]
```

Locks are used to synchronize the actions of asynchronous processes. When a process sets a lock, that process has exclusive access to the data, or whatever, controlled by the lock. Clearing a lock via the UNLOCK statement, described below, allows other processes to set the lock so that they can access the data controlled by the lock.

If the WAIT qualifier is specified, the process issuing the LOCK statement will wait until the lock is cleared and then set it. If the SET qualifier is specified, the process does not wait if another process has the lock set. The boolean variable is set to TRUE if the LOCK statement was able to set the lock, and FALSE otherwise.

The testing and setting of the lock occurs as a single non-interruptable operation by using the hardware's "compare swap" instruction.

6.12.5 UNLOCK (*)

The UNLOCK statement is used to clear a lock set via the LOCK statement as described above.

```
<unlock> ::= UNLOCK <sp> <lock variable>
```

6.13 Deleting Variables

The DELETE_VARIABLE command can be used to delete procedure and environment variables. It is described in the section on control commands.

7 Expressions

An expression is used to designate or compute a value of a particular type.

```
<expression> ::= <structure expression>
               | <elemental expression>
               | <application expression>
```

7.1 Elemental Expressions

```
<elemental expression> ::= <boolean expression>
                         | <COBOL name expression>
                         | <data name expression>
                         | <file expression>
                         | <integer expression>
                         | <keyword expression>
                         | <lock expression> (*)
                         | <name expression>
                         | <program name expression>
                         | <real expression>
                         | <statistic code expression>
                         | <status code expression>
                         | <string expression>
                         | <string pattern expression> (*)
```

7.1.1 Boolean Expression

```
<boolean expression> ::= <boolean term 1> [<boolean sum | diff> <boolean term 1>]...

<boolean sum | diff> ::= <boolean sum> | <boolean difference>

<boolean sum> ::= <sp> OR <sp>

<boolean difference> ::= <sp> XOR <sp>

<boolean term 1> ::= <boolean term 2> [<boolean product> <boolean term 2>]...

<boolean product> ::= <sp> AND <sp>

<boolean term 2> ::= [<boolean complement>] <boolean operand>

<boolean complement> ::= NOT <sp>

<boolean operand> ::= <boolean variable>
                    | <boolean function>
                    | <boolean>
                    | <relational expression>
                    | <(> <boolean expression> <)>

<relational expression> ::= <relational term> [<relation> <relational term>]

<relation> ::= <equal to>
             | <not equal to>
             | <greater than>
             | <greater than or equal to>
             | <less than>
             | <less than or equal to>
```

```
<equal to> ::= [<sp>] = [<sp>]

<not equal to> ::= [<sp>] <> [<sp>]

<greater than> ::= [<sp>] > [<sp>]

<greater than or equal to> ::= [<sp>] >= [<sp>]

<less than> ::= [<sp>] < [<sp>]

<less than or equal to> ::= [<sp>] <= [<sp>]

<relational term> ::= <expression>
```

Because of the use of spaces as separators, the boolean sum, difference, product and complement operators may not be used in an expression for a command or function parameter or list element. Functions are provided that perform these operations and they can be used in any boolean expression. For the same reason spaces may not be used around the relational operators in these situations.

Care must be excersized in the use of the "=" relational operator in an expression for a command parameter. An expression of the form "name=something" given positonally for a boolean parameter is interpreted as an attempt to give a parameter called "name" the value "something." Such an expression must be given either by specifying the parameter by name, e.g. "param=name=something," or by parenthesizing the expression, e.g. (name=something).

Boolean expressions are evaluated from left to right according to the precedence of the operators defined in the above syntax definition.

The result of a boolean sum (OR) is TRUE if either of its operands is TRUE, and FALSE otherwise. If its left operand is TRUE, the right operand is not evaluated.

The result of a boolean difference (XOR) is TRUE if one but not both of its operands is TRUE, and FALSE otherwise.

The result of a boolean product (AND) is TRUE if both of its operands are TRUE, and FALSE otherwise. If its left operand is FALSE, the right operand is not evaluated.

The result of a boolean complement (NOT) is TRUE if its operand is FALSE, and FALSE otherwise.

The result of a relation operator is TRUE if its operands satisfy the relation designated by the operator according to the comparison rules described below, and FALSE otherwise. The relational operators that test for equality, "=," and inequality, "<>," of their operands can be used with any data type except "application." The remaining relational operators require that some kind of ordering be defined for their operands, and therefore can only be used with some data types.

The order of the operands of a relational operator is significant in that the type of the left operand determines how the right operand is evaluated. Therefore when comparing a variable or parameter with something make the variable or parameter the left operand and the something the right operand.

7.1.1.1 Comparison of Strings

Strings are compared by comparing corresponding characters, starting with the first (leftmost) characters, until two such characters are unequal or all characters have been compared. If unequal characters are encounterred the result of the comparison is determined by the relative position of those characters in the ASCII (American Standard Code for Information Interchange) character set. If one string is shorter than the other it is treated as if it were extended on the right with space characters to the length of the longer string.

7.1.1.2 Comparison of Names, Keywords, Data_Names and COBOL_Names

When comparisons are made that involve a name, keyword or COBOL_name the corresponding operand is treated as if it were a string consisting of the characters that comprise the name, keyword or COBOL_name with lowercase letters in the name treated as the corresponding uppercase letters. If a string literal is compared with a name any lowercase letters in it are treated as the corresponding uppercase letters also.

7.1.1.3 Comparison of Program_Names

When comparisons are made that involve a program_name the corresponding operand is treated as if it were a string consisting of the characters that comprise the program_name. Unlike comparisons involving other forms of names, the case of letters is significant when comparing program_names.

7.1.1.4 Comparison of File (Path) Names

When comparisons are made that involve a path name the corresponding operand is treated as if it were a string consisting of the characters that comprise the path name with lowercase letters in the path name treated as the corresponding uppercase letters. If a name or keyword is compared with a path name, the name is treated as relative path (see the section on file expressions).

7.1.1.5 Summary of Comparisons of Elemental Types

The table below describes what combinations of operands of the various elemental types may be compared.

Left Operand	Right Operand	Notes
boolean	boolean	False is less than true.
COBOL_name	COBOL_name	See Comparison of Names.
COBOL_name	string literal	
data_name	data_name	See Comparison of Names.
data_name	keyword	
data_name	name	
data_name	string literal	
file	file	See Comparison of File Names.
integer	integer	
integer	real	
keyword	keyword	See Comparison of Names.
keyword	data_name	
keyword	name	
keyword	string literal	
lock	lock	Locks may only be compared for equality and inequality.
name	name	See Comparison of Names.
name	data_name	
name	keyword	
name	string literal	
program_name	program_name	See Comparison of Program
program_name	COBOL_name	Names.
program_name	data_name	
program_name	name	
program_name	string literal	
real	real	
real	integer	
statistic_code	statistic_code	
status_code	status_code	
string	string	See Comparison of Strings.
string_pattern	string_pattern	String_patterns may only be compared for equality and inequality.

7.1.1.6 Comparison of Application Values

Application values may not be compared.

7.1.1.7 Comparison of Arrays

Arrays may only be compared if their element types and bounds are the same. Only the equality and inequality relations may be used with arrays. Two arrays are equal if their corresponding elements are equal.

7.1.1.8 Comparison of Command Reference Values

See “Comparison of Records.”

7.1.1.9 Comparison of Date_Time Values

Date_time values are compared by comparing the corresponding numerical components in the order year, month, day, hour, minute, second and millisecond in turn until an inequality is found or all components have been compared.

7.1.1.10 Comparison of Entry_Point_References

See “Comparison of Records.”

7.1.1.11 Comparison of Line_Identifiers

See “Comparison of Records.”

7.1.1.12 Comparison of Lists

Lists may only be compared if their element types are the same. Only the equality and inequality relations may be used with lists. Two lists are equal if they have the same number of elements and their corresponding elements are equal.

7.1.1.13 Comparison of Ranges

Ranges may only be compared if their element types are the same. Only the equality and inequality relations may be used with ranges. Two ranges are equal if their corresponding low and high elements are equal.

7.1.1.14 Comparison of Records

Only records of the same type may be compared and only for equality or inequality. Two records are equal if their corresponding fields are equal.

7.1.1.15 Comparison of Status Values

See “Comparison of Records.”

7.1.1.16 Comparison of Time_Increment Values

Time_increment values are compared by comparing the corresponding numerical components in the order years, months, days, hours, minutes, seconds and milliseconds in turn until an inequality is found or all components have been compared.

7.1.1.17 Comparison of Time_Zone Values

See “Comparison of Records.”

7.1.1.18 Comparison of Types

Values that are type specifications can only be compared for equality and inequality.

7.1.2 COBOL Name Expression

A “COBOL name” is a name whose syntax is acceptable in the programming language COBOL but does not conform to the SCL syntax for names.

A COBOL name consists of from 1 to 30 letters, digits and hyphens, the first and last of which may not be hyphens and one of which must be a letter.

```
<COBOL name expression> ::= <COBOL name variable>
                          | <COBOL name function>
                          | <COBOL name>

<COBOL name> ::= <letter> [[<COBOL name char>]... <letter | digit>]
               | <digit> [<COBOL name char>]... <letter>
                 [[<COBOL name char>]... <letter | digit>]

<COBOL name char> ::= <letter | digit> | <hyphen>

<letter | digit> ::= <letter> | <digit>

<hyphen> ::= -
```

7.1.3 Data Name Expression

```
<data name expression> ::= <name>
                         | <data name variable>
                         | <data name function>
                         | <name variable>
                         | <name function>
                         | <keyword variable>
                         | <keyword function>
                         | <data name pattern>

<data name pattern> ::= <name pattern>
```

The evaluation of a data name expression differs from that of a name expression in that if an unqualified name is given—even if it is the name of a function, variable or parameter—the name itself becomes the value of the expression. The evaluation of an unqualified name can be forced via the $NAME function.

Data names containing “wild card” characters (wild card patterns) can only be used in data name expressions that are part of list expressions. They provide a convenient notation for referencing groups of data names that adhere to a “naming convention.” See the section on name expressions for more information on how wild cards can be used with lists of names.

7.1.4 File Expression

```
<file expression> ::= <file reference>
                    | <file>
                    | <catalog>

<file | catalog> ::= <file> | <catalog>

<file reference> ::= <path> [<cycle or file position>]...

<//> ::= //

<file> ::= <path> [<cycle reference specification>]...

<catalog> ::= <path>

<path> ::= <file variable> [<relative path specification>]
```

```
         | <file function> [<relative path specification>]
         | <absolute path>
         | <relative path>

<absolute path> ::= <command file path>
                  | <deferred path>
                  | <family path>
                  | <generic path>
                  | <system file>
                  | <user path>

<command file path> ::= <:> $COMMAND [<.> <command file qualifier>]
                      | <:> $COMMAND_OF_CALLER

<command file qualifier> ::= <name>

<deferred path> ::= <deferred path header> [<relative path specification>]

<deferred path header> := <:> $DEFER <.> <file variable name>
                        | <:> $DEFER <.> <file function name>

<family path> ::= <:> <family name> [<relative path specification>]

<generic path> ::= <:> <generic path name> [<relative path specification>]

<generic path name> ::= $FAMILY | $LOCAL | $PROCEDURE | $SOURCE | $SOURCE_OF_CALLER | $SYSTEM
                      | $TASK | $UP | $USER | $UTILITY | $WORKING_CATALOG | $WC

<system file> ::= <job file>
                | <standard file>

<job file> ::= COMMAND | INPUT | OUTPUT
             | $JOB_LOG | $JOB_MESSAGE (*) | $NULL

<standard file> ::= $ECHO | $ERRORS | $INPUT | $LIST | $OUTPUT | $RESPONSE

<user path> ::= <.> <user name> <relative path specification>

<relative path specification> ::= <//> <relative path expression>
                                | <.> <relative path>

<relative path expression> ::= <path part expression> <relative path specification>

<path part expression> ::= <name variable>
                         | <name function>
                         | <keyword variable>
                         | <keyword function>
                         | <data_name variable>
                         | <data_name function>

<relative path> ::= <path element> [<relative path specification>]

<path element> ::= <generic path element>
                 | <path reversal>
                 | <path traversal>
                 | <path element name>
                 | <path element name pattern>

<generic path element> ::= $USER

<path element name pattern> ::= <wild card pattern>

<path reversal> ::= $UP

<path traversal> ::= $ALL

<cycle or file position> ::= <cycle reference specification>
```

```
                            | <file position specification>

<cycle reference specification> ::= <//> <cycle reference expression>
                                  | <.> <cycle reference>

<cycle reference expression> ::= <cycle number expression>
                               | <cycle reference variable>
                               | <cycle reference function>

<cycle number expression> ::= <generic cycle reference> [<+|-> <unsigned decimal>]
                            | <cycle number variable> [<+|-> <unsigned decimal>]
                            | <cycle number function> [<+|-> <unsigned decimal>]

<generic cycle reference> ::= $HIGH | $LOW | $NEXT

<cycle number variable> ::= <integer variable>

<cycle number function> ::= <integer function>

<cycle reference variable> ::= <integer variable>
                             | <name variable>

<cycle reference function> ::= <integer function>
                             | <name function>

<cycle reference> ::= <cycle number>
                    | <generic cycle reference>
                    | <all cycles>

<cycle number> ::= <unsigned decimal>

<all cycles> ::= $ALL

<file position specification> ::= <//> <file position expression>
                                | <.> <file position>

<file position expression> ::= <file position variable>
                             | <file position function>
                             | <file position>

<file position variable> ::= <name variable>

<file position function> ::= <name function>

<file position> ::= $ASIS | $BOI | $EOI
```

The result of the evaluation of a file expression is always an absolute path optionally containing a cycle reference and/or file position. The result may contain generic elements which are resolved when the file or catalog designated by the expression is accessed. Thus there are two stages to the processing of a file expression. The first stage, evaluation, consists of determining the values of any variables or functions in the expression and constructing an absolute path following the rules described below. The second stage, resolution, occurs when the file expression is being used to access a file or catalog, or upon request; and consists of interpreting any generic path elements or cycle references.

For convenience, more than one file position may be present in a file expression; the result contains only the rightmost one. Also, more than one cycle reference may be present in a file expression; the result contains only the rightmost one. All file positions and cycle references must appear at the end of a file expression, although they may be intermixed in any order.

When a file expression is evaluated a concatenation operator (//) is replaced by a period and the value of the following <path part expression>, <cycle reference expression> or <file position expression> is used in the resulting path. These “expressions” are normally simply references to a variable or function. A file expression is evaluated from left to right.

For a <user path>, the family name used is that of the current user.

The <job file>s and <standard file>s are considered to be in the catalog that is local to the current job, i.e. :$LOCAL These files may only ever have one cycle (cycle 1).

If a relative path is specified for a file expression it is automatically converted to an absolute path, being considered to be

relative to the working catalog.

If a file expression is to be specified as a relative path and the first name in the actual relative path happens to be the name of a file variable or function, the evaluation of that variable or function can be inhibited by prefixing the relative path with the $WORKING_CATALOG ($WC) function which explicitly designates the working catalog.

There are a number of functions provided that return various commonly used absolute paths as their values. See the section on functions for more information.

The use of a <cycle reference expression> is evaluated to a <cycle number>. A cycle number must be in the range 1..65535. $HIGH represents the number of the currently highest cycle of a file and may only be used with a catalogued file. $NEXT is equivalent to $HIGH+1 for a catalogued file and 1 (one) for a new file. $LOW represents the number of the currently lowest cycle of a file and may only be used with a catalogued file.

If no cycle is specified for a file, the cycle used is dependent upon the application (command) in which the reference occurs. In the vast majority of cases $HIGH is assumed for a catalogued file, and 1 (one) for a new file. Some commands, however, such as delete_file, assume $LOW.

Paths containing wild card characters (path element name patterns) can only be used in file expressions that are part of a list expression. See the subsections on wild card patterns within the sections on string pattern expressions and list expressions for details on the form and meaning of wild card patterns.

Each catalogued file or catalog that adheres to the naming convention specified by the wild card characters becomes an element of the resulting list.

A wild card pattern can be used for any particular level in the file hierarchy. When the number of subcatalogs appearing in a path is unknown or varies, the “path traversal” notation, i.e. $ALL, must be used. $ALL behaves very similarly to the wild card multiple in a wild card pattern. Specifically it matches a sequence of zero or more path elements in a file hierarchy.

When $ALL is used at the end of a path, its meaning depends on the path element that immediately precedes it. If that element designates a file, $ALL matches all of the cycles of the file. If the preceding element is a catalog, $ALL matches all paths contained in the catalog, i.e. all cycles of all files in the catalog plus all subcatalogs and the files and subcatalogs they contain, and so on down the hierarchy.

7.1.4.1 Generic Paths

A <command file path>, <deferred path> or <generic path> can be used to postpone the resolution of a path until it is used to access a file or catalog. The names that are first in a <command file path> or <generic path> are the same as the names of certain functions. The difference between using the path notation (beginning with a colon) and the function notation (beginning with a name) is that the function is fully evaluated at the time of file expression evaluation, whereas the path notation is not resolved until the file or catalog is accessed.

The meanings of the various generic names are:

$COMMAND	designates the SCL interpreter's current source of input. (The value returned by the $COMMAND function is a path of the form :$COMMAND.xxx where “xxx” designates the particular source of input at the time the $COMMAND function was called.)
$COMMAND_OF_CALLER	is intended for use within an SCL procedure and designates the $COMMAND file of the caller of the procedure.
$DEFER	is used to defer the evaluation of a file variable or function. It must be followed by the name of the variable or function to be evaluated when the path is resolved.
$FAMILY	designates the family name of the user on whose behalf the job is running.
$LOCAL	designates the catalog of temporary files local to the job.
$PROCEDURE (*)	designates the catalog of temporary files local to a call to an SCL procedure.
$SOURCE	is intended for use within an SCL procedure or a program description and designates the command list entry in which the procedure or program description resides. When used as a <generic path name>, $SOURCE must refer to an object library or catalog.

$SOURCE_OF_CALLER	is similar to $SOURCE but designates the source of the caller of a procedure or program description.
$SYSTEM	designates the name of the system family or the system user for the mainframe. The name of the system family may differ from mainframe to mainframe, however the system family for the current mainframe may be generically referred to as :$SYSTEM. The name of the system "user" is always $SYSTEM. (The $SYSTEM function returns a path of the form :$SYSTEM.$SYSTEM, i.e. the generic path for the system user on the current mainframe.)
$TASK (*)	designates the catalog of temporary files local to a task.
$UP	designates <path reversal>, i.e. rather than designating an element at the next level down in the catalog hierarchy, it has the effect of going up one level. It can be thought of as removing the currently rightmost element of the path, or reversing the normal left to right evaluation. $UP is processed during the evaluation stage of a file expression if the element that precedes it is not generic, otherwise its effect is delayed until the resolution stage. $UP is particularly useful following a generic path like $WORKING_CATALOG or $SOURCE. :$UP is equivalent to :$WORKING_CATALOG.$UP.
$USER	designates the user name of the user on whose behalf the job is running. :$USER is equivalent to :$FAMILY.$USER.
$UTILITY (*)	designates the catalog of temporary files local to a call to a command utility.
$WORKING_CATALOG, $WC	designates the current working catalog.

7.1.5 Integer Expression

```
<integer expression> ::= <numeric expression>
```

An integer expression is a numeric expression with an integer result. If the result of the expression as given is of type real, an error results.

7.1.6 Keyword Expression

```
<keyword expression> ::= <keyword>
                       | <keyword variable>
                       | <keyword function>
                       | <name variable>
                       | <name function>
                       | <data name variable>
                       | <data name function>
                       | <keyword pattern>

<keyword pattern ::= <name pattern>
```

The evaluation of a keyword expression differs from that of a name expression in that an unqualified name given for the expression is treated as a reference to a variable or function only if it is not one of the keywords for the particular type,

The result of the keyword expression is the "nominal" name for the keyword given, i.e. the first keyword within the group in which the given keyword was defined (see the section on definition of keyword types for more information).

Keywords containing "wild card" characters (wild card patterns) can only be used in keyword expressions that are part of list expressions. They provide a convenient notation for referencing groups of keywords that adhere to a "naming convention." All keywords defined for the keyword type are candidates against which the wild card pattern is matched.; however, only "nominal" keywords result from the expansion of a keyword pattern.

7.1.7 Lock Expression (*)

```
<lock expression> ::= <lock variable>
                    | <lock function>
```

7.1.8 Name Expression

```
<name expression> ::= <name variable>
                    | <name function>
                    | <keyword variable>
                    | <keyword function>
                    | <data name variable>
                    | <data name function>
                    | <name>
                    | <name pattern>

<name pattern> ::= <wild card pattern>
```

A name given for a name expression is first “looked up” to see if it is the name of a name variable or name function. (This implies that names for variables of type name should be chosen very carefully to avoid confusion.) If it becomes necessary to inhibit this lookup, the $NAME function, which converts a string to a name, can be used.

Names containing “wild card” characters (wild card patterns) can only be used in name expressions that are part of list expressions. They provide a convenient notation for referencing groups of names that adhere to a “naming convention.”

For file expressions, the catalogs in the file system hierarchy can be interrogated to obtain the list of candidate names to which to apply the name pattern. In lieu of an analogous mechanism for name expressions, the following convention has been devised. The “type” of the expression, which is generically “list of name,” must be given a “type name.” This type name can be defined via an explicit type declaration or as a parameter attribute when the parameters for a command or function are declared. The list of name expression evaluator prefixes a $ (dollar sign) character to this type name, if necessary, to obtain the name of an SCL function and then calls that function to obtain the list of candidate names.

All of this is only done if wild card characters are used in the expression. If any step fails, i.e. if the type does not have a name, or prefixing that name with a $ does not yield the name of a function, or the function has required parameters or does not return a list of names, the matching process is considered to have failed just as though all of these steps had succeeded but no names that matched the pattern were found.

An application can also declare the list with the DEFER_EXPANSION attribute. This means that SCL will not “expand” the wild card references, but pass on the wild card pattern as an APPLICATION value. This allows the application to more efficiently provide wild card facilities to its users.

It cannot be emphasized strongly enough that any application that deals with lists of names shouldtake advantage of the DEFER_EXPANSION option when declaring the list type, or at least provide functions and type names so that users of the application can take advantage of the convenience provided by the wild card pattern notation.

7.1.9 Numeric Expression

```
<numeric expression> ::= <numeric term 1> [<add | sub> <numeric term 1>]...

<add | sub> ::= <add> | <subtract>

<add> ::= [<sp>] + [<sp>]

<subtract> ::= [<sp>] - [<sp>]

<numeric term 1> ::= <numeric term 2> [<multiply | divide> <numeric term 2>]...

<multiply | divide> ::= <multiply> | <divide>

<multiply> ::= [<sp>] * [<sp>]
```

```
<divide> ::= [<sp>] / [<sp>]

<numeric term 2> ::= <numeric term 3> [<exponentiate> <numeric term 3>]...

<exponentiate> ::= [<sp>] ** [<sp>]

<numeric term 3> ::= [<plus | minus>] <numeric operand>

<plus | minus> ::= <plus> | <minus>

<plus> ::= + [<sp>]

<minus> ::= - [<sp>]

<numeric operand> ::= $MAX
                    | $MIN
                    | <integer operand>
                    | <real operand>
                    | <(> <numeric expression> <)>

<integer operand> ::= <integer variable>
                    | <integer function>
                    | <unsigned integer>

<real operand> ::= <real variable>
                 | <real function>
                 | <unsigned real>
```

Numeric expressions are used to compute integer or real results.

In an expression for a command or function parameter or list element spaces may not be used around the operators.

In the absence of parentheses, the numeric operators are evaluated from left to right except for the exponentiation operator which is evaluated from right to left.

The result of evaluating an operator is an integer if both operands are integers and the result can be represented as an integer; and is real otherwise. The result of division involving two integer operands, however, is always an integer.

In a context where either an integer or real number result is acceptable and the actual result of the expression can be represented as an integer, the evaluation of the expression will yield an integer, even if there were real number operands or intermediate results in the expression.

$MAX and $MIN are special numeric operands. They are used to represent the maximum and minimum values, respectively, in which an integer or real expression may result.

7.1.10 Program Name Expression

A program name designates an entry point or module name of an executable program. A program name may have the form of a standard SCL name, a COBOL name or, if the syntax of the program name doesn't conform to either of those, a string of from 1 to 31 characters. In the string form the case of letters is significant, i.e. folding of lowercase letters to uppercase does nottake place as it does for SCL and COBOL names.

```
<program name expression> ::= <program name variable>
                            | <program name function>
                            | <name expression>
                            | <COBOL name expression>
                            | <string expression>
                            | <program name>
                            | <program name pattern>

<program name> ::= <name>
                 | <COBOL name>
                 | <string>
```

```
<program name pattern> ::= <name pattern>
```

Program names containing “wild card” characters (wild card patterns) can only be used in program name expressions that are part of list expressions. They provide a convenient notation for referencing groups of program names that adhere to a “naming convention” and conform to the syntax of SCL names. See the section on name expressions for more information on how wild cards can be used with lists of names.

7.1.11 Real Expression

```
<real expression> ::= <numeric expression>
```

A real expression is a numeric expression with a real result. If the result of the expression as given would be of type integer, the result is automatically converted to a real number.

7.1.12 Statistic Code Expression

```
<statistic code expression> ::= <statistic code variable>
                              | <statistic code function>
                              | <statistic code name>
                              | <statistic code string>
                              | <integer>
```

A <statistic code name> is a name whose form is one of the following:

xxnnn

xx_nnn

where xx is the statistic code's two character “product identifier,” and nnn is the statistic code's number.

A <statistic code string> is a <string> whose form is:

'XX nnn'

where XX the statistic code's two character “product identifier,” and nnn is the statistic code's number. (Lower case letters in XX are folded to their upper case counterparts.)

7.1.13 Status Code Expression

```
<status code expression> ::= <status code variable>
                           | <status code function>
                           | <status code name>
                           | <status code string>
                           | <integer>
```

A <status code name> is either a name whose form is one of the following:

xxnnn

xx_nnn

(where xx is the status code's two character “product identifier,” and nnn is the status code's number), or a name defined along with the status code's message template and severity. In the latter case the name's form is usually one of the following:

xxe$aaaaa

xxe#aaaaa

A <status code string> is a <string> whose form is:

'XX nnn'

where XX the status code's two character “product identifier,” and nnn is the status code's number. (Lower case letters in XX are folded to their upper case counterparts.)

7.1.14 String Expression

```
<string expression> ::= <string operand> [<concatenate> <expression>]...

<concatenate> ::= [<sp>] // [<sp>]

<string operand> ::= <string variable>
                   | <string function>
                   | <string>
                   | <(> <string expression> <)>

<string literal expression> ::= <string>
```

The concatenation operator sticks two strings together, e.g.

'abc' // 'def' = 'abcdef'

If the string type controlling the evaluation of the expression included the literal qualifier (i.e. STRING LITERAL), the <string literal expression> syntax is used rather than the <string expression> syntax.

In an expression for a command or function parameter, or list element, spaces may not be used around the concatenate operator.

If the right operand of a concatenation operator is not a string it is converted to a string if possible and the result of the conversion is used. (The conversion is not possible if more than one string is required to represent the value of the expression.) This ability of the concatenation operator to implicitly convert data to its string representation often avoids the need to use functions such as $string, $integer_string, etc.

7.1.15 String Pattern Expression

A string pattern expression is used to determine whether a string contains a “pattern” of characters. A string pattern may be very simple, i.e. a string literal; or very complex, involving alternatives, concatenations and patterns produced by the use of SCL provided functions.

```
<string pattern expression> ::= <string pattern operand>
                                [<concatenate> <string pattern operand>]...

<string pattern operand> ::= <string pattern variable>
                           | <string pattern function>
                           | <string operand>
                           | <(> <string pattern expression> <)>
```

When a <string operand> is used it is implicitly converted to a string pattern that matches the string.

In pattern matching the concatenation operator asks the question, “does the string contain this pattern followed by that pattern?”

Functions are provided that produce a number of patterns useful for various purposes or combine patterns to form more complex ones. See the section “Functions Related to String Patterns” for more information.

String pattern expressions are evaluated from left to right according to the precedence of the operators defined in the above syntax definition.

7.1.15.1 “Wild Card” Patterns

A wild card pattern is a concise, albeit cryptic, way of specifying a useful subset of string patterns. A wild card pattern consists of a series of literal characters, and wild card specifiers. The wild card specifiers are described below.

A wild card pattern can be used in the following situations.

1. For a path element in a file expression that is a component of a “list of file” expression.
2. For an element expression that is a component of a “list of name” expression. (Also, “list of data_name,” “list of keyword,” and “list of program_name.)
3. As a parameter of various functions.

```
<wild card pattern> ::= <wild card basic pattern>
                      | <wild card extended pattern>

<wild card basic pattern> ::= <wild card basic element>...

<wild card extended pattern> ::= <wild card extended element>...

<wild card basic element> ::= <wild card basic char>
                            | <wild card single>
                            | <wild card multiple>
                            | <wild card quoted string>

<wild card basic char> ::= <any ascii character except ?, *, or '>

<wild card extended element> ::= <wild card extended char>
                               | <wild card single>
                               | <wild card multiple>
                               | <wild card quoted string>
                               | <wild card class>
                               | <wild card inverse class>
                               | <wild card alternation>

<wild card extended char> ::=
          <any ascii character except ?, *, ', [, ], { or }>

<wild card single> ::= ?

<wild card multiple> ::= *

<wild card quoted string> ::= ' [<any ascii character except '>]... '

<wild card class> ::= [ <wild card class component>... ]

<wild card inverse class> ::= [ ^ <wild card class component>... ]

<wild card class component> ::= <wild card class single>
                              | <wild card class group>
                              | <wild card class range>

<wild card class single> ::= <any ascii character except ', [ or ]>
                           | <wild card quoted char>

<wild card quoted char> ::= ' [<any ascii character except '>] '

<wild card class group> ::= <wild card quoted string>

<wild card class range> ::= <wild card class single> -
                            <wild card class single>

<wild card alternation> ::= { <wild card alternative>
                          [ | <wild card alternative> ]... }

<wild card alternative> ::= <wild card pattern>
                          | <empty>
```

When wild card patterns appear in file references or names, the SCL option `wild_card_pattern_type` determines

whether the basic or extended type of pattern is used. The functions that accept wild card patterns for matching strings allow the pattern type to be specified. This option is provided because the bracket characters may be used in SCL names, so there must be a way specify them literally. When the basic pattern type is used, the characters that are special in extended patterns have no special significance. The semantics of the wild card pattern elements are described below.

? This character (<wild card single>) matches any single character.

* This character (<wild card multiple>) matches any string of characters, including a zero-length (null) string.

' This character is used to enclose other special characters that are to be matched literally. Two ' characters in succession means a single '.

Note: the <wild card quoted string> and <wild card quoted char>, described above, may not be used in a wild card pattern for a file or name.

[ ] This notation (<wild card class>) can be used in an extended pattern and matches any of the characters enclosed by the square brackets. If either of the characters [or] are to be members of the class, they must be enclosed in ' characters. If a ' character is to be a member of the the class, two consecutive ' characters must be used.

The characters ? and * characters have no special significance within a character class pattern element. The - character has no special significance if it is the first or last <wild card class component>.

A shorthand notation for representing a range of characters that are to be members of the class is provided. This notation consists of placing a - between the characters at the low and high ends of the range. All of the ASCII characters between and including the characters on either side of the - are included in the class, regardless of the collating order of those characters (i.e. [0-9] and [9-0] are equivalent and include all the decimal digits).

For example:

```
[cmptv]
```

matches any of the characters “c”, “m”, “p”, “t” and “v”,

```
[$a-z#]  or       [#$z-a]
```

matches a “$”, any letter or a “#”.

[^] This notation (<wild card inverse class>) is very similar to that above except that the characters between the square brackets following the first ^ character specify characters that are *not* to be members of the class. The ^ character has no special significance except when it immediately follows the [of the class.

For example:

```
[^0-9]
```

matches any character that is not a decimal digit.

{ } This notation (<wild card alternation>) can be used in an extended pattern and matches any one of a number of alternative sub-patterns. The alternatives are separated from one another by the | character. An <empty> alternative matches the null string. The | character has no special significance outside of a <wild card alternation>.

For example:

```
{Donna|John|Robyn}
```

matches “Donna”, “John” or “Robyn”.

7.2 Structure Expressions

```
<structure expression> ::= <array expression>
                         | <command reference expression>
```

```
                          | <date time expression>
                          | <entry point reference expression>
                          | <line identifier expression>
                          | <list expression>
                          | <range expression>
                          | <record expression>
                          | <status expression>
                          | <time increment expression>
                          | <time zone expression>
```

7.2.1 Array Expression

```
<array expression> ::= <array variable>
                     | <array function>
                     | <(> <expression> [<,|sp> <expression>]... <)>
```

An array expression given as a sequence of expressions enclosed in parentheses provides an expression for each element of the array. The results of these expressions are assigned to the elements of the array from its lower_bound to its upper_bound. When an entire array is being assigned to a variable or parameter, a value for every element must be specified.

7.2.2 Command Reference Expression

```
<command reference expression> ::= <command reference variable>
                                 | <command reference function>
                                 | <command reference>
```

7.2.3 Date Time Expression

```
<date time expression> ::= <date time variable or function> [<add | sub> <time amount>]...
                         | <date> [<.> <time>]
                         | <time>
                         | <date time string>

<date time variable or function> ::= <date time variable>
                                   | <date time function>

<time amount> ::= <time increment variable or function>

<time increment variable or function> ::= <time increment variable>
                                        | <time increment function>

<date> ::= [<year>] - [<month>] - [<day>]

<time> ::= [<hour>] <:> [<minute>] [<:> [<second>] [<.> [<millisecond>]]]

<date time string> ::= <string>

<year> ::= <unsigned decimal>
         | <(> <integer expression> <)>

<month> ::= <unsigned decimal>
          | <(> <integer expression> <)>

<day> ::= <unsigned decimal>
        | <(> <integer expression> <)>

<hour> ::= <unsigned decimal>
         | <(> <integer expression> <)>
```

```
<minute> ::= <unsigned decimal>
           | <(> <integer expression> <)>

<second> ::= <unsigned decimal>
           | <(> <integer expression> <)>

<millisecond> ::= <unsigned decimal>
                | <(> <integer expression> <)>
```

The date time notation provides a standard way to express a date and/or time. The date alone, the time alone, or both may be given; but the resulting value includes both. If the date is omitted, today's date is used. If the time is omitted, all time components are set to zero. If the year, month, day, hour or minute is omitted, the corresponding component of today's date or the present time is used. If the second or millisecond is omitted, zero is used.

7.2.3.1 Date and Time Formats

NOS/VE supports a number of ways of writing date and time values in addition to the standard form supported for date_time expressions. If a date_time value is given in the form of an SCL string literal, the contents of that string are interpreted as a date_time value by matching the string against a number of "date time form strings."

A "date time form string" specifies the order and form of the components of a date time representation and the separators between those components. The notations described in the following table are used within "date time form strings."

Notation	Description
Y2	two digit year (00..99)
Y4	four digit year (1900..2155)
MN(xxx)	name of the month in natural language xxx—if (xxx) is not specified, the currently selected natural language is used
MA(xxx)	abbreviation for the name of the month in natural language xxx—if (xxx) is not specified, the currently selected natural language is used
M2	two digit number of the month (01..12)
D2	two digit day of the month (01..31)
J3	three digit day of the year (001..366)
DN(xxx)	name of the day of the week in natural language xxx—if (xxx) is not specified, the currently selected natural language is used
DA(xxx)	abbreviation for the name of the day of the week in natural language xxx—if (xxx) is not specified, the currently selected natural language is used
MONTH	a “shorthand” equivalent of the date form string “MN D2, Y4” (this is the released system default date format)
MDY	a “shorthand” equivalent of the date form string “M2/D2/Y2”
DMY	a “shorthand” equivalent of the date form string “D2.M2.Y2”
ISOD	a “shorthand” equivalent of the date form string “Y4-M2-D2”
ORDINAL	a “shorthand” equivalent of the date form string “Y4J3”
H24	two digit hours in the “24-hour” clock form (00..23)
H12	one or two digit hours in the “12-hour” clock form (1..12)
AMORPM	two character “half of the day” indicator (AM or PM)
MM	two digit minutes (00..59)
SS	two digit seconds (00..59)
S10	one digit tenths of a second (0..9)
S100	two digit hundredths of a second (00..99)
S1000	three digit thousandths of a second (000..999)
AMPM	a “shorthand” equivalent of the time form string “H12:MM AMORPM” (this is the released system default time format)
HMS	a “shorthand” equivalent of the time form string “H24:MM:SS”
MILLISECOND, MS	a “shorthand” equivalent of the time form string “H24:MM:SS.S1000”
ISOT	a “shorthand” equivalent of the time form string “H24.MM.SS,S100”
TZ(xxx)	identifier of the system's time zone in natural language xxx—if (xxx) is not specified, the currently selected natural language is used
TZA(xxx)	abbreviation for the identifier of the system's time zone in natural language xxx—if (xxx) is not specified, the currently selected natural language is used

A “date time form string” is composed from these notations combined with delimiters chosen from the characters: “ ” (space), “,” (comma), “/” (slant), “-” (hyphen) and “.” (dot).

When a date is converted to its string representation leading zeros are omitted from the day of the month number in any format involving one of the MN or MA month formats immediately followed by a “ ” (space) or a “-” (hyphen). Leading zeros are provided in all other cases.

A comma in the string representation is always followed by a space unless the item following the comma is S10, S100 or S1000.

When interpreting a <date time string> leading zeros may be omitted from any component number provided the component is preceded by a delimiter or letter.

When interpreting a <date time string> or a “date time form string” contiguous spaces are treated as a single space, and spaces on either side of a comma are ignored.

Interpretation of a <date time string> may occur in one of two ways: under control of a “date time form string” or not. In the context of a <date time expression> only the latter possibility exits. Both possibilities exist, however, when the $date_time function (described in a later section of this document) or the clp$convert_string_to_date_time program interface (described in the NOS/VE Program Interface ERS) are used.

In the absence of a “date time form string” interpretation of a <date time string> consists of matching the contents of the

string with one of the “date time form strings” shown in the table below. The table shows form strings for the date and time formats separately. These parts may appear alone or any order in the <date time string>. If both the date and time appear in the <date time string> they must be separated from each other by one of the delimiters mentioned above.

In the presence of a “date time form string” interpretation of a <date time string> consists of examining the string guided by the information contained in the “date time form string.” In this method a day name (one of the DN or DA forms) may appear in the <date time string>, if allowed by the “date time form string”; however, the day name will be ignored as far as producing the resuling date_time value is concerned.

Date Form	Form Name	Example
'MN D2, Y4'	MONTH	November 1, 1984
'MA D2, Y4'		Nov 1, 1984
'D2 MN Y4'		1 November 1984
'D2 MA Y4'		1 Nov 1984
'D2 MN Y2'		1 November 84
'D2 MA Y2'		1 Nov 84
'D2-MN-Y4'		1-November-1984
'D2-MA-Y4'		1-Nov-1984
'D2-MN-Y2'		1-November-84
'D2-MA-Y2'		1-Nov-84
'D2MNY4'		01November1984
'D2MAY4'		01Nov1984
'D2MNY2'		01November84
'D2MAY2'		01Nov84
'M2/D2/Y4'		11/01/1984
'M2/D2/Y2'	MDY	11/01/84
'Y4-J3'		1984-306
'Y4J3'	ORDINAL	1984306
'Y2-J3'		84-306
'Y2J3'		84306
'Y4-M2-D2'	ISOD	1984-11-01
'Y4M2D2'		19841101
'Y2-M2-D2'		84-11-01
'Y2M2D2'		841101
'D2.M2.Y4'		01.11.1984
'D2.M2.Y2'	DMY	01.11.84
Time Form	**Form Name**	**Example**
'H12:MM AMORPM'	AMPM	2:41 PM
'H24:MM:SS'	HMS	14:41:38
'H24:MM:SS.S1000'	MILLISECOND, MS	14:41:38.629
'H24.MM.SS,S100'	ISOT	14.41.38,62

7.2.3.2 Month and Day Name Modules

NOS/VE supports naming of months and days in natural languages other than US_ENGLISH or ENGLISH. The support for such naming in US_ENGLISH and ENGLISH is built-in but must be “added on” for other languages. The mechanism for adding such support for other languages is the message module (see the Object Code Management section for information on the generation and use of message modules).

The conventions for defining a message module containing the names of the months and days in a natural language other than US_ENGLISH are as follows:

- The “seed” name for the message module is MONTHS_AND_DAYS. (Thus the name of the module for the French language would be MONTHS_AND_DAYS$FRENCH.)
- The “parameter prompt” message is used to define the name for each month and day. Use as the “parameter name” the name of the month or day in English. Use as the “message template” text in the following format:

```
<full name> [<,> <abbreviated name>]
```

If the abbreviated name is omitted, the first three characters of the full name are used.

- The module must contain specifications for all of the months of the year and days of the week.

7.2.3.3 Time Zone Identifier Modules

NOS/VE supports identifying (naming) of time zones in any natural language. The mechanism for providing identifiers for time zones is the message module (see the Object Code Management section for information on the generation and use of message modules).

The conventions for defining a message module containing time zone identifiers are as follows:

- The “seed” name for the message module is TIME_ZONES. (Thus the name of the module for the US_English language is TIME_ZONES$US_ENGLISH.)
- The “parameter prompt” message is used to define the identifier for a time zone.
- Use as the “parameter name” a name with the following format:

  <st or dst> $ [_] <hours digits> [$ [_] <minutes digits>]

  where <st or dst> is STANDARD_TIME or DAYLIGHT_SAVING_TIME, <hours digits> represents the hours_from_gmt component of the time zone, and <minutes digits> represents the minutes_offset component of the time zone. If the time zone's hours_from_gmt is negative an underscore (_) must precede the <hours digits>. If the time zone's minutes_offset is negative an underscore (_) must precede the <minutes digits>. The <minutes digits> should only be included in the name if the time zone's minutes_offset is not zero.
- Use as the “message template” text in the following format:

  ```
  <full identifier> [<,> <abbreviated identifier>]
  ```

  If the abbreviated identifier is omitted, it is formed from the first characters of each word of the <full identifier>.

```
Example: create_message_module time_zones$us_english
         create_parameter_prompt_message standard_time$0
         Greenwich Mean Time
         **
         creppm standard_time$_6
         Central Standard Time
         **
         creppm daylight_saving_time$_6
         Central Daylight Saving Time, CDT
         **
         end_message_module
```

7.2.4 Entry Point Reference Expression

```
<entry point reference expression> ::= <entry point reference variable>
                                     | <entry point reference function>
                                     | <file> <//> <program name variable>
                                     | <file> <//> <program name function>
                                     | <file> <.> <name>
                                     | <file> <.> <string>
                                     | <program name variable>
                                     | <program name function>
                                     | <program name>
```

7.2.5 Line Identifier Expression

A line identifier is to designate a line of text in a Source Code Utility (SCU) “deck.”

```
<line identifier expression> ::= <line identifier>
                                 | <line identifier variable>
                                 | <line identifier function>

<line identifier> ::= <modification name> <.> <sequence number>

<sequence number> ::= <unsigned decimal>
```

A modification name may not contain more than 9 (nine) characters.

A sequence number must be in the range 1..0ffffff(16)

See the SCU ERS, DCS# ARH3883, for more details on the meaning and usage of line identifiers.

7.2.6 List Expression

```
<list expression> ::= <list variable>
                    | <list function>
                    | <(> <expression> [<,|sp> <expression>]... <)>
                    | <elemental expression>
                    | <array variable>
                    | <array function>

<list rest expression> ::= <list variable>
                         | <list function>
                         | <expression> [<,|sp> <expression>]...
```

When a list expression is specified as a sequence of expressions enclosed in parentheses, the result of each such expression becomes the value of the corresponding element of the list. The number of expressions in the sequence determines the size of the list.

If a single value (elemental expression) is given for a list, the single value is automatically converted into a list.

If an array value is supplied for a list, the array value is automatically converted to a list.

When the type being used to direct the evaluation of the list has the REST qualifier, the <list rest expression> syntax is employed. In this case the parentheses enclosing the sequence of expressions for the list elements may not be given explicitly; they are implied by the context of the <list rest expression>.

This form of the list type may only be used for the last parameter of a function or the last field of a record. In both instances, the closing parenthesis of the parameter/field list is taken to be the end of the list rest expression. If a value is being assigned to a record field that is of a LIST REST type, i.e. the assignment is done individually rather than as part of an expression for the entire record, the <list expression> syntax is employed.

7.2.6.1 “Wild Card” Patterns as List Generators

For lists whose element type is file, name, program_name, data_name or keyword, the so called “wild card” characters can be used as a shorthand notation for designating groups of names or path names that adhere to a “naming convention.”

Wild card patterns are fully described in the subsection devoted to them in the section on string pattern expressions.

See the sections on file expressions and name expressions for information on how the candidate names for the matching operation are obtained.

It is always considered to be an error if none of the candidate names or path names matches a specified pattern.

7.2.6.2 Path Traversal as a List Generator

The “path traversal” and “all cycles” notation, i.e. $ALL, can be used in a file expression that is part of a “list of file” expression. This notation is used to represent a section of a file hierarchy that spans more than one level, or to represent all cycles of a file. It is fully described in the section on file expressions.

7.2.7 Range Expression

```
<range expression> ::= <range variable>
                     | <range function>
                     | <(> <expression> [<..> <expression>] >)>
                     | <expression> [<..> <expression>]
```

The first expression specifies the “low” value of the range and the second expression specifies the “high” value. The terminology of “low” and “high” should not be interpreted to mean that the evaluation of a range expression will verify that the “low” value is less than or equal to the “high” value. If such validation is required it must be supplied by the command or function processor that deals with the range value. The terms “low” and “high” mean nothing more than “the value to the left of the ellipsis” and “the value to the right of the ellipsis,” respectively.

If the ellipsis and second expression are omitted, both the “low” and “high” values of the range are set to the value of the expression given.

7.2.8 Record Expression

```
<record expression> ::= <record variable>
                      | <record function>
                      | <(> [<expression>] [<,|sp> [<expression>]]... <)>
                      | <elemental expression>
```

A record expression given as a sequence of expressions enclosed in parentheses provides an expression for each field of the record. The results of these expressions are assigned to the fields of the record in the order in which the fields of the record type were defined. If an expression for a field is omitted, the corresponding field is uninitialized. A field may only be omitted if its <field requirement> is $OPTIONAL.

If a single value (elemental expression) is given for a record, the single value is assigned to the first field of the record. All remaining fields of the record must have <field requirement>s of $OPTIONAL and will be uninitialized.

7.2.9 Status Expression

```
<status expression> ::= <status variable>
                      | <status function>
```

7.2.10 Time Increment Expression

```
<time increment expression> ::= <time amount> [<add | sub> <time amount>]...
                              | <time zone variable or function> <sub>
                                <time zone variable or function>
                              | <date time variable or function> <sub>
                                <date time variable or function>
                              | <date increment> [<.> <time increment>]
                              | <time increment>

<time zone variable or function> ::= <time zone variable>
                                   | <time zone function>

<date increment> ::= [<years>] - [<months>] - [<days>]

<time increment> ::= [<hours>] <:> [<minutes>] [<:> [<seconds>] [<.> [<milliseconds>]]]

<years> ::= <unsigned decimal>
          | <(> <integer expression> <)>

<months> ::= <unsigned decimal>
```

```
           | <(> <integer expression> <)>

<days> ::= <unsigned decimal>
         | <(> <integer expression> <)>

<hours> ::= <unsigned decimal>
          | <(> <integer expression> <)>

<minutes> ::= <unsigned decimal>
            | <(> <integer expression> <)>

<seconds> ::= <unsigned decimal>
            | <(> <integer expression> <)>

<milliseconds> ::= <unsigned decimal>
                 | <(> <integer expression> <)>
```

Although the time increment notation is similar to the date time notation, its meaning is different. Rather than a specific instant of time it represents an amount of time.

7.2.11 Time Zone Expression

```
<time zone expression> ::= <time zone variable>
                         | <time zone function>
                         | <time zone>

<time zone> ::= <hours from gmt> [<:> <minutes offset>] [<.> <daylight saving time>]

<hours from gmt> ::= [<+|->] <unsigned decimal>
                   | <(> <integer expression> <)>

<minutes offset> ::= [<+|->] <unsigned decimal>
                   | <(> <integer expression> <)>

<daylight saving time> ::= DAYLIGHT_SAVING_TIME | DST
                         | STANDARD_TIME | ST
                         | <(> <boolean expression> <)>
```

If a <boolean expression> is given for <daylight saving time>, TRUE is equivalent to using DAYLIGHT_SAVING_TIME and FALSE is equivalent to STANDARD_TIME.

Although the time zone notation is similar to the time increment notation, its meaning is different. Rather than an arbitrary amount of time it represents a difference in time from Greenwich Mean Time (GMT).

7.3 Application Expression

The following syntax description for an “application expression” is presented to illustrate the constraints imposed by the SCL interpreter on the form such an expression can have. Stated in words, what the syntax description says is that the application expression must be balanced with respect to parentheses, apostrophes and quote marks.

```
<application expression> ::= <application variable>
                           | <application function>
                           | [.] [<av element> [.]]...

<av element> ::= ( <nested av element> <)|eol>
               | ' <any ascii character other than '> <'|eol>
               | " <any ascii character other than "> <"|eol>
               | <unnested av char>

<)|eol> ::= ) | <end of line>
```

```
<'|eol> ::= ' | <end of line>
<"|eol> ::= " | <end of line>

<unnested av char> ::= <any ascii character other than ().,; or <space>>

<nested av element> ::= <av element> | <space> | , | ; | .
```

If the specification for the APPLICATION type being used to evaluate the expression included the BALANCE_BRACKETS attribute, the “[” and “{” characters are treated identically to the “(” character, and the “]” and “}” characters are treated identically to the “)” character.

8 Command and Function Procedures

Command and function processors can be written in the System Command Language as well as in the standard NOS/VE supported programming languages, most notably CYBIL and FORTRAN. The descriptions in the following subsections are oriented to command and function processors written in SCL, however all of what is said about the specifications for parameters and help modules applies equally well to commands otherwise implemented. See the NOS/VE Program Interface ERS for a description of the GENERATE_PDT tool that accepts as input "procedure header" and produces as output the CYBIL variable declarations to hold the parameter description table (PDT) it corresponds to.

An SCL command procedure may be packaged on an object library or on a file all by itself. In the former case it may be called by any of the procedure names contained in its definition, whereas in the latter case it may only be called by the name of the file in which it resides.

An SCL function procedure may only be packaged on an object library.

The definition of both SCL command and function procedures consists of two main parts, a header in which the names for the command or function are defined and a specification for the parameters of the command or function.

The next two subsections describe the details of command and function procedures respectively. These are followed by a subsection for each of the major characteristics that command and function procedures have in common.

8.1 Command Procedures

```
<command procedure> ::= <command procedure header>
                        [<command procedure body>]
                        <command procedure end>

<command procedure header> ::= <bol> PROCEDURE
                               <procedure attributes spec>
                               <command names>
                               [[<sp>] <(> [<command parameter definitions>] <)>] <;|nl>
                               [[<check statement>] <;|nl>]...

<procedure attributes spec> ::= [<sp>] <(> <procedure attributes> <)> [<sp>]
                              | <sp>

<procedure attributes> ::= <procedure attribute> [<,|sp> <procedure attribute>]...

<procedure attribute> ::= <help module reference>
                        | <procedure scope attribute>
                        | ADVANCED
                        | HIDDEN

<help module reference> ::= <name>

<procedure scope attribute> ::= GATE | LOCAL | XDCL

<command names> ::= <command name> [<,|sp> <command name>]...

<command parameter definitions> ::= <command parameter definition>
                                    [<;|nl> <command parameter definition>]...

<command parameter definition> ::= <parameter names> [
                                   [<sp|nl>] <:> [<sp|nl>]
                                   [<(> <command parameter attributes> <)> [<sp|nl>]]
                                   <type expression>]
                                   [[<sp|nl>] = [<sp|nl>] <parameter default specification>]

<parameter names> ::= <parameter name> [<,|sp> <parameter name>]...

<command parameter attributes> ::= <command parameter attribute>
                                   [<,|sp> <command parameter attribute>]...
```

```
<command parameter attribute> ::= <parameter mode>
                                | ADVANCED
                                | BY_NAME
                                | CHECK
                                | DEFER
                                | HIDDEN
                                | SECURE
                                | <type name>

<parameter mode> ::= VAR

<parameter default specification> ::= $REQUIRED
                                    | $OPTIONAL
                                    | [$CONFIRM <sp|nl>] [<default name> <,> [<nl>]]
                                      <expression>

<command procedure body> ::= <statement list>

<command procedure end> ::= PROCEND [<sp> <command name>] <eol> | <eop> | <eoi>
```

The command names and procedure attributes are only meaningful when the procedure resides on an object library.

If the BY_NAME parameter attribute is specified for a parameter, that parameter, if given on the command call, must be given by name. If the BY_NAME attribute is omitted, the parameter may be given positionally as well as by name. Parameters for which positional significance is meaningless or undesirable should be be given the BY_NAME attribute. For example, all status parameters should have the BY_NAME attribute.

The default type for a command parameter is FILE unless the name of the parameter is STATUS. A command parameter named STATUS has default attributes of BY_NAME and VAR, and a default type of STATUS.

If a command name is given following PROCEND it must match the first command name given in the procedure header.

8.2 Function Procedures

```
<function procedure> ::= <function procedure header>
                         [<function procedure body>]
                         <function procedure end>

<function procedure header> ::= <bol> FUNCTION
                                <procedure attributes spec>
                                <function names>
                                [[<sp>] <(> [<function parameter definitions>] <)>] <;|nl>
                                [[<check statement>] <;|nl>]...

<function names> ::= <function name> [<,|sp> <function name>]...

<function parameter definitions> ::= <function parameter definition>
                                     [<;|nl> <function parameter definition>]...

<function parameter definition> ::= <parameter name> [<sp|nl>] <:> [<sp|nl>]
                                    [<(> <function parameter attributes> <)> [<sp|nl>]]
                                    <type expression>
                                    [[<sp|nl>] = [<sp|nl>] <parameter default specification>]

<function parameter attributes> ::= <function parameter attribute>
                                    [<,|sp> <function parameter attribute>]...

<function parameter attribute> ::= ADVANCED
                                 | CHECK
                                 | DEFER
                                 | HIDDEN
                                 | <type name>
```

```
<function procedure body> ::= <statement list>

<function procedure end> ::= FUNCEND [<sp> <function name>] <eol> | <eop> | <eoi>
```

All function names must begin with a dollar sign ($) character.

The definition of parameters for a function differs somewhat from that for a command. Only one parameter name may be specified for a function parameter. This is because the parameter name may not be used on a call to the function since all function parameters must be given positionally. The parameter name only serves as the means of referencing the parameter value within the body of the function. Also, the type specification for a function parameter may not be omitted. This is because, although the most common type of command parameter is a file there is no typical function parameter.

If a function name is given following FUNCEND it must match the first function name given in the procedure header.

A function procedure establishes the rturned value of the function by executing an EXIT statement with a WITH clause. See the section on the EXIT statement for more information.

8.3 Procedure Scope Attributes

The scope attributes for commands and functions are defined as follows.

GATE (*) The GATE attribute signifies that the procedure can be invoked from a ring outside its execution bracket (above or below). The execution bracket of a procedure is determined by the ring attributes of the file on which it resides. The execution bracket consists of the range of rings between and including the R1 and R2 ring attribute values. The GATE attribute permits a procedure to be called from its call bracket in addition to just its execute bracket. The call bracket for a procedure is the range of rings from 3 (three) to the R3 ring attribute of the file containing the procedure. If a procedure is called from below its R1 ring, it will execute in its R1 ring. If a procedure is called from above its R2 ring, it will execute in its R2 ring. If a procedure is called from between and including its R1 and R2 rings, it will execute in the ring in which it is called. The GATE attribute implies the XDCL attribute.

LOCAL (*) The LOCAL scope attribute signifies that the procedure may only be called by a procedure or program that was invoked from the same object library.

XDCL The XDCL scope attribute signifies that the procedure may be called from outside the object library on which it resides as well as from within that library.

The default procedure scope attribute is XDCL.

8.4 Help Module References

The help module name designates where the help and prompt information for the command can be found. The help module is used in a LINE or SCREEN mode parameter dialog for a command; and also by the DISPLAY_COMMAND_INFORMATION and DISPLAY_FUNCTION_INFORMATION commands. It contains information about how to use the command or function and its parameters.

If the help module reference is omitted, the last of the command or function names defined in the procedure header is used.

The name actually used to search for the help module is the name specified or defaulted suffixed with a dollar sign ($) character followed by the name of the preferred natural language selected for the job. The object libraries in the command list are searched for a help module of the resulting name. If no such module is found and the preferred natural language is not US_English, the search is repeated using US_English as the natural language.

If no help module is found help information and prompts are generated from the information contained in the procedure header. Procedures that reside outside of object libraries are treated as though they do not have help modules.

8.5 Advanced Commands and Functions

If the ADVANCED attribute is specified for a command or function, information about the it is not shown, e.g. by the DISPLAY_COMMAND_LIST_ENTRY command, unless requested.

8.6 Hidden Commands and Functions

If the HIDDEN attribute is specified for a command or function, information about the it is not shown, e.g. by the DISPLAY_COMMAND_LIST_ENTRY command.

8.7 Parameter Passing Modes

The parameter mode attribute determines how the parameter is passed to the command or function. If VAR is specified the "value" passed to the parameter must be a variable; this allows the command to write the variable as well as use its value. If VAR is omitted the parameter can be given as any expression of the appropriate type and can only be read by the command or function.

The VAR attribute may not be specified for a function parameter or for a parameter with the SECURE attribute.

Specifying VAR as the parameter passing mode affects the interpretation of the type specification for the parameter. If the type is a qualified union type, then a variable whose type is one of the union's member types may be passed to the parameter. For example, given the following:

```
TYPE
  some_name: ANY OF
       COBOL_NAME
       NAME
     ANYEND
TYPEND

VAR
  p: some_name
  c: COBOL_NAME
  n: NAME
VAREND

PROCEDURE example (
  module, m: (VAR) some_name
  status)
  "...
PROCEND
```

all of the following calls are allowed:

```
example p
example c
example n
```

8.8 Deferred Evaluation of Parameters

If the DEFER attribute is specified for a parameter, the evaluation of an expression assigned to it is deferred until the parameter is referenced, and is reevaluated each time it is referenced. The DEFER attribute cannot be used in combination with the VAR attribute.

8.9 Advanced Parameters

If the ADVANCED attribute is specified for a parameter, that parameter is prompted for in a LINE or SCREEN mode parameter dialog, only if requested. Information about the parameter is shown only if requested. Command parameters with the ADVANCED attribute are automatically given the BY_NAME attribute.

8.10 Hidden Parameters

If the HIDDEN attribute is specified for a parameter, that parameter is never prompted for in a parameter dialog. Information about the parameter is never advertised. Command parameters with the HIDDEN attribute are automatically given the BY_NAME attribute.

8.11 Secure Parameters

Specifying the SECURE attribute for a parameter has two effects. First, if the parameter is prompted for in a parameter dialog, the prompting includes terminal input/output conventions that enable private entry of the parameter.

Second, a command procedure with one or more SECURE parameters that resides on an object library is given the “manually log” attribute, meaning it is the command processor's responsibility to cause calls to the command to be logged and/or echoed. See the section on Command Log Options, below, for more information.

A parameter with the SECURE attribute may not have the VAR attribute, and may not be a function parameter.

8.12 The Type Name Parameter Attribute

A name can be given for the type of a parameter when the parameter is defined. See the section on name expressions for a discussion of how this type name is used in conjunction with a parameter whose generic type is “list of name” (*).

8.13 Default Values for Parameters

The parameter default specification determines whether a parameter must be given on a call to the command, or can be omitted and if so what its default value is, if any. If no default specification is given, $OPTIONAL is assumed.

$REQUIRED specifies that the parameter must be supplied when the command is called. A parameter with the ADVANCED or HIDDEN attribute may not also have the $REQUIRED attribute.

$OPTIONAL specifies that the parameter may be omitted when the command is called, and if omitted no default value is assigned to the parameter.

The $CONFIRM attribute is only meaningful when the command is called interactively in “line” mode. It specifies that if the parameter is omitted, its default value must be confirmed. A parameter with the ADVANCED or HIDDEN attribute may not also have the $CONFIRM attribute.

A <default name> specifies a name optionally defined via the CREATE_DEFAULT_VARIABLE command. If at the time the command is called the name is defined, its associated string is interpreted as the expression for the default value. If the name is not defined the default specification's expression is used.

If an expression is given in the default specification, the parameter may be omitted and if omitted the expression is assigned as the parameter's value.

8.14 Extended Parameter Checking

Checking for validity beyond that resulting from information specified in the definition of the parameter can be performed by parameter statements within command and function procedures.

For example, a parameter can be defined to be of type name and the SCL interpreter can validate that only a name is given for the parameter value. However, if the name must be one of a particular set of names (e.g. the name of a module in an object library), it is up to the command processor to validate that only an appropriate name is supplied as the parameter value.

The CHECK/CHECKEND statement is provided to facilitate performing this level of parameter validation. Errors reported via an EXIT CHECK statement within a CHECK/CHECKEND statement can be corrected by a user of the command or function in a parameter dialog just as if the errors had been detected by the SCL interpreter directly. See the section on Interactive Usage for information about parameter dialogs.

To a large extent, use of this facility eliminates from the user's vantage point effects related to “who” detects errors. This

is of primary importance to the processing of commands via parameter dialogs. Also, this facility provides for treating an error as a “command_fault” rather than an “execution_fault” when the “fault” lies with the call to the command. See the section on Conditions and Condition Handlers for information about command and execution faults.

8.14.1 The CHECK Parameter Attribute

Specifying the CHECK attribute for a parameter indicates that extended checking is to be performed for the parameter. A CHECK/CHECKEND statement should be provided in this case.

If a CHECK/CHECKEND statement is defined for the parameter, the CHECK attribute is implicitly selected.

(The CHECK attribute is primarily intended for use when defining a parameter description table (PDT) for a command or function processor implemented in CYBIL. Selecting the CHECK attribute causes the processor supplied “check parameters procedure,” if any, to be called for the parameter.)

8.14.2 CHECK / CHECKEND (*)

The CHECK/CHECKEND statement is used to bracket statements that perform parameter validation specific to a command or function procedure that is over and above the validation possible by the SCL interpreter.

There should be a CHECK/CHECKEND statement for each parameter the procedure will validate. Such a statement might perform “semantic” (as opposed to syntactic) validation (e.g. is the value given for a parameter meaningful in the context of the procedure). There could also be a CHECK/CHECKEND statement for the procedure “as a whole.” Such a statement might be responsible for checking parameter interdependancies.

If an error is detected by a CHECK/CHECKEND statement, it reports the problem using the EXIT CHECK statement specifying the error via the WITH clause.

The CHECK/CHECKEND statements are executed after each user interaction with SCL's parameter evaluation mechanism, i.e. when that mechanism has performed all of the validation it can. If the parameter to be validated by a particular CHECK/CHECKEND statement has not been changed since the last time the statement was executed, the statement is skipped. On the first execution of CHECK/CHECKEND statements after the procedure call, all parameters would be treated as “changed.”

No other statements may be used in a procedure between the procedure header and the first CHECK/CHECKEND statement or between CHECK/CHECKEND statements.

```
<check statement> ::= CHECK [<sp> <parameter name>] <;|nl>
                        [<statement list>]
                      CHECKEND [<sp> <parameter name>]
```

If a <parameter name> is specified at the beginning of a CHECK/CHECKEND statement, the statement is assumed to contain validation for that parameter only. In this case the same <parameter name> may be given on the CHECKEND clause as a way of increasing the readability of the procedure.

8.15 Command Log Options

The responsibility for logging and/or echoing the call to a command is determined by the “command log option” selected for the command.

By having the “automatically log” option selected for it, a command processor is releived of the responsibility for logging and/or echoing; the responsibility rests with the SCL interpreter. This option will cause the interpreter to perform the appropriate action just prior to invoking the processor for the command.

By having the “manually log” option selected for it, a command processor is given the responsibility for logging and/or echoing. This option is established for those command procedures that have at at least one parameter with the SECURE attribute. Commands processors implemented other than as SCL procedures must have this attribute explicitly defined for them, either via the command table for utility subcommands or via CREATE_OBJECT_LIBRARY subcommands for commands that reside on object libraries. Commands that reside directly in a catalog cannot be given the manual log option.

A command procedure containing a CHECK/CHECKEND statement will have its manual logging performed upon exit from that statement. A command procedure without a CHECK/CHECKEND statement will have manual logging performed upon completion of the evaluation of its parameters. For a command processor implemented in CYBIL, the interface that induces manual logging to take place is clp$call_complete; see section 2 of the NOS/VE Program Interface ERS for details.

Whether any logging or echoing actually occurs, automatically or manualy, is determined by the origin of the command call as described in the follwoing paragraphs.

A command is logged if it originates from the main COMMAND file of the job. Command logging consists of writing the command to the job log and, if “system logging” has been activated, to the system log.

A command is a candidate for being echoed if it originates from any file to which the user has read permission, or if it originates from the processing of an INCLUDE_COMMAND or INCLUDE_LINE command in which echoing was enabled. A command which is a candidate for echoing is actually echoed if the $ECHO file is connected to one or more files.

9 9 Funtions

This section describes many of the functions that are built in to the SCL interpreter. These functions are grouped according to the kind of object (data type, variable, file, etc.) they operate upon or otherwise deal with. Some functions are generic in the sense that they deal with more than one type of data. For example, the $size function is used to determine the number of elements in a list and the number of characters in a string. Such generic functions are described in the subsection for each group to which they apply. Each description discusses the function as if it belonged only to that group. This style of description is used to make clear the use of each function as it applies to a particular group.

Not all functions are described in this section. Those that are not are described in the sections that talk about the area to which the function is related.

9.1 Function Calls

```
<function> ::= <function name> <(> [<function parameters>] <)>
               [<parenthesized value qualifier> [<value qualifier>]...]
             | <function name> [<(> [<function parameters>] <)>]
               [<field qualifier> [<value qualifier>]...]

<function parameters> ::= <function parameter> [<,|sp> <function parameter>]...

<function parameter> ::= <expression>
```

A function call consists of the name of the function optionally followed by a series of parameters enclosed in parentheses. If there are no parameters to be passed to the function the parentheses may be omitted.

A function call may also be followed by value qualifiers (see the section on variable references). A left parenthesis immediately following a function name is always interpreted as the opening delimiter of the function's parameter list; so if a subscript or substring qualifier is used with a function to which no parameters are passed, it is necessary to supply an empty parameter list.

9.2 Functions Related to Arrays

9.2.1 $array

The $ARRAY function converts a list to an array.

```
$array (
      list: list = $required
      )
```

LIST: This parameter specifies the list to be converted.

9.2.2 $lower_bound

The $LOWER_BOUND function returns the lower_bound (lowest possible subscript value) of an array.

```
$lower_bound (
      array: array = $required
      )
```

ARRAY: This parameter specifies the array whose lower bound is to be returned.

9.2.3 $upper_bound

The $UPPER_BOUND function returns the upper_bound (highest possible subscript value) of an array.

```
$upper_bound (
      array: array = $required
      )
```

ARRAY: This parameter specifies the array whose upper bound is to be returned.

9.3 Functions Related to Booleans

These functions are provided so that non-trivial boolean expressions may be passed as parameters to commands and functions. Because of the use of spaces as parameter separators, it is not possible to use expressions involving the mnemonic operators (AND, NOT, OR and XOR) directly in expressions for parameters.

9.3.1 $and (*)

The $AND function returns a boolean value of TRUE if the value of all of its boolean operands are TRUE, and FALSE otherwise.

```
$and (
      operands: (defer) list 2..$max_list of boolean = $required
      )
```

OPERANDS: This parameter specifies the boolean expressions to be AND'd together. If any operand evaluates to FALSE, the remaining operands are not evaluated.

9.3.2 $boolean (*)

The $BOOLEAN function returns the boolean equivalent of its parameter.

```
$boolean (
      source: any of
          string
          integer
        anyend = $required
      )
```

SOURCE: This parameter specifies the value to be converted to a boolean value. If a string is given, it must contain a boolean constant. If an integer is given, FALSE is returned for zero, and TRUE is returned for any other value.

9.3.3 $if

The $IF function returns one of two results, selected according to the value of a boolean expression. It is particularly useful in functions such as $BUILD_RESULT.

```
$if (
      condition: boolean = $required
      result_if_true: (defer) any = $required
      result_if_false: (defer) any = $required
      )
```

CONDITION: This parameter specifies the boolean expression.

RESULT_IF_TRUE: This parameter specifies the value to be returned if CONDITION is true.

RESULT_IF_FALSE: This parameter specifies the value to be returned if CONDITION is false.

9.3.4 $not

The $NOT function returns the boolean complement of its boolean parameter.

```
$not (
      value: boolean = $required
      )
```

VALUE: This parameter specifies the value to be complemented.

9.3.5 $or (*)

The $OR function returns a boolean value of TRUE if the value of any of its boolean parameters is TRUE, and FALSE otherwise.

```
$or (
      operands: (defer) list 2..$max_list of boolean = $required
      )
```

OPERANDS: This parameter specifies the boolean expressions to be OR'd together. If any operand evaluates to TRUE, the remaining operands are not evaluated.

9.3.6 $xor (*)

The $XOR function returns a boolean value of TRUE if one, but not both, of its boolean parameters is true.

```
$xor (
      first: boolean = $required
      second: boolean = $required
      )
```

FIRST: This parameter specifies the first operand.

SECOND: This parameter specifies the second operand.

9.4 Functions Related to COBOL Names

9.4.1 $cobol_name (*)

The $COBOL_NAME function interprets the contents of a string as a COBOL name.

```
$cobol_name (
      string: string = $required
      )
```

STRING: This parameter specifies the string to be converted.

9.5 Functions Related to Command and Function Processing

9.5.1 $caller_interactive (*)

The $CALLER_INTERACTIVE function returns a boolean value of TRUE if the SCL procedure using the function was called directly from an interactive terminal, and FALSE otherwise. It has no parameters.

```
$caller_interactive (
      )
```

9.5.2 $caller_ring (*)

The $CALLER_RING function returns an integer value that designates the ring of the caller of the SCL procedure that calls the function. If it is not called from within an SCL procedure it returns the current ring of execution (i.e. is equivalent to the $RING function). It has no parameters.

```
$caller_ring (
      )
```

9.5.3 $command

The $COMMAND function returns a file reference to the input stream that the SCL interpreter is currently reading from. It has no parameters.

```
$command (
      )
```

$COMMAND must be used with care. Ultimately it should only be used on the INCLUDE_FILE command, although it may passed through any number of layers of procedure call before such use. This is because $COMMAND represents more than the file itself; it also represents the line and character position within the file. In fact it may not represent a file at all, but a “line” introduced by the INCLUDE_LINE command. The SCL interpreter is the only part of NOS/VE that knows how to access the data designated by the value returned by $COMMAND.

9.5.4 $command_of_caller

The $COMMAND_OF_CALLER function is similar to the $COMMAND function but is intended for use within SCL procedures. The file reference it returns designates the input stream that contained the call to the procedure within which the function is used. This is necessary since in the same context $COMMAND would return the a file reference to the procedure itself. It has no parameters.

```
$command_of_caller (
      )
```

The same cautions described for the $COMMAND function also apply to the use of the $COMMAND_OF_CALLER function.

9.5.5 $expected_type (*)

The $EXPECTED_TYPE function is intended for use by a function procedure and returns the specification for the type of value the SCL expression evaluator expects the function to return. It has no parameters.

```
$expected_type (
      )
```

An example of a function that uses this capability is the $APPLY function which tailors its return value according to the data type it is expected to return.

9.5.6 $interaction_style

The $INTERACTION_STYLE function is used to determine what the current interaction style is. The type of value it returns depends on the parameter passed to it..

```
$interaction_style (
      option: key
          style
          (menu_rows, menu_row, mr)
          (extend_utility_interaction, eui)
          (line, l)
          (screen, s)
          (desktop, dt, d)
        keyend = style
      )
```

OPTION: This parameter selects the component of the interaction style to return.

STYLE: This option causes a keyword value of LINE, SCREEN or DESKTOP to be returned, representing the current interaction style.

MENU_ROWS, MENU_ROW, MR: This option causes the integer number of default menu rows to be returned.

EXTEND_UTILITY_INTERACTION, EUI: This option will cause a boolean value of TRUE to be returned if extended utility interaction has been enabled, and FALSE otherwise.

LINE, L: This options causes a boolean value of TRUE to be returned if the current interaction style is LINE, and FALSE otherwise.

SCREEN, S: This options causes a boolean value of TRUE to be returned if the current interaction style is SCREEN, and FALSE otherwise.

DESKTOP, DT, D: This options causes a boolean value of TRUE to be returned if the current interaction style is DESKTOP, and FALSE otherwise.

Omission causes STYLE to be used.

9.5.7 $message_level

The $message_level function is used to determine what the current message level is.

```
$message_level (
      level: key
          (brief, b)
          (full, f)
        keyend = $required
      )
```

LEVEL: This parameter specifies which message level is being interrogated.

BRIEF, B: This options causes a boolean value of TRUE to be returned if the current message level is BRIEF, and FALSE otherwise.

FULL, F: This options causes a boolean value of TRUE to be returned if the current message level is FULL, and FALSE otherwise.

9.5.8 $natural_language

The $natural_language function returns the name of the preferred natural language for the job. It has no parameters.

```
$natural_language (
      )
```

9.6 Functions Related to Command References

9.6.1 $command_reference (*)

The $command_reference function interprets the contents of a string as a command reference.

```
$command_reference (
      string: string = $required
      )
```

STRING: This parameter specifies the string to be converted.

9.7 Functions Related to Dates, Times, Time Increments, and Time Zones

9.7.1 $date

The $DATE function returns a string representing the date portion of a date_time value. The size of the returned string depends on the format selected as a parameter for the function.

```
$date (
      format: any of
          key
            default
            month
            mdy
            dmy
            (isod, iso)
            ordinal
          keyend
          string
        anyend = default
      date: date = $now
      )
```

FORMAT: This parameter specifies the format of the returned string as a “date form string,” described in the subsection “Date Time Formats” in the section on “Date Time Expressions.” A keyword value may be specified for one of the date formats. The keywords correspond to the identically spelled “date time form string notations, except for the keyword DEFAULT. DEFAULT designates the site definable default date format. The released system value of this default is MONTH.

Omission causes DEFAULT to be used.

DATE: This parameter specifies the date value to be converted.

Omission causes the current date ($NOW) to be used.

```
Example: display_value $date(month)
         November 1, 1984
         display_value $time(hms)//' on '//$date(isod)
```

```
          14:41:38 on 1984-11-01
```

9.7.2 $date_time

The $DATE_TIME function interprets the contents of a string as a date_time value.

```
$date_time (
      string: string = $required
      format: string
      )
```

STRING: This parameter specifies the string to be converted.

FORMAT: This parameter specifies the format of the returned string as a “date_time form string,” described in the subsection “Date Time Formats” in the section on “Date Time Expressions.”

Omission causes the rules used to evaluate a <date time string> in the context of a date time expression to be used.

9.7.3 $date_time_string

The $DATE_TIME_STRING function returns a string representing a date_time value. The size of the returned string depends on the format selected as a parameter for the function.

```
$date_time_string
      format: any of
          key
            default
            ampm
            dmy
            hms
            (isod, iso)
            isot
            mdy
            (millisecond, ms)
            month
            ordinal
          keyend
          string
        anyend = default
      date_time: date_time = $now
      )
```

FORMAT: This parameter specifies the format of the returned string as a “date_time form string,” described in the subsection “Date Time Formats” in the section on “Date Time Expressions.” In addition, a keyword value may be specified for one of the date or time formats. The keywords correspond to the identically spelled “date time form string notations, except for the keyword DEFAULT. DEFAULT designates the site definable default date format followed by a dot followed by the site definable default time format. The released system value of default is MONTH for the date format and HMS for the time format.

Omission causes DEFAULT to be used.

DATE_TIME: This parameter specifies the date value to be converted.

Omission causes the current date_time ($NOW) to be used.

```
Example: display_value $date_time_string('month. ampm')
          November 1, 1984. 14:41:38
```

9.7.4 $day

The $DAY function returns a string representing the day of the week corresponding to a date_time value.

```
$day (
      format: any of
          key
            (full, dn, f)
            (brief, da, b)
          keyend
          string
        anyend = full
      date: date = $now
      )
```

FORMAT: This parameter specifies the format of the returned string as a “day form string,” described in the subsection “Date Time Formats” in the section on “Date Time Expressions.” In addition, a keyword value may be specified for one of the day formats; these keywords and their equivalent “day of the week form strings” are shown in the following table.

Keyword	Day Form
FULL, DN, F	'DN'
BRIEF, DA, B	'DA'

Omission of the format parameter causes the full name of the day in the current natural language to be used.

DATE: This parameter specifies the date value to be converted.

Omission causes the current date ($NOW) to be used.

```
Example: display_value $date('ma d2, y4')
         Nov 1, 1984
         display_value $day
         Tuesday
```

9.7.5 $local_date_time

The $LOCAL_DATE_TIME function returns the “local” date and time equivalent to a date and time expressed relative to the “universal” time zone. The “universal” time zone is that of Greenwich, England, often called Greenwich Mean Time.

```
$local_date_time (
      universal_date_time: date_time = $required
      time_zone: time_zone = $required
      )
```

UNIVERSAL_DATE_TIME: This parameter specifies the “universal” date_time value to be converted.

TIME_ZONE: This parameter specifies the “local” time_zone.

9.7.6 $now

The $NOW function returns the date_time value for the current date and time, i.e. the date and time it is now, relative to the system's time zone. It has no parameters.

```
$now (
      )
```

9.7.7 $time

The $TIME function returns a string representing the time portion of a date_time value. The time is returned in a format specified as a parameter to the function.

```
$time (
      format: any of
          key
            default
            ampm
            hms
            isot
            (millisecond, ms)
          keyend
          string
        anyend = default
      time: time = $now
      )
```

FORMAT: This parameter specifies the format of the returned string as a “time form string,” described in the subsection “Date Time Formats” in the section on “Date Time Expressions.” In addition a keyword value may be specified for one of the time formats. The keywords correspond to the identically spelled “date time form string” notations, except for the keyword DEFAULT. DEFAULT designates the site definable default time format. The released system value for this default is AMPM.

Omission of the format parameter causes DEFAULT to be used.

TIME: This parameter specifies the time value to be converted.

Omission causes the current time ($NOW) to be used.

```
Example: display_value 'Time is now '//$time(ampm)
         Time is now 2:41 PM
         display_value 'Time is now '//$time(hms)
         Time is now 14:41:38
```

9.7.8 $time_zone

The $TIME_ZONE function returns the system's time zone. It has no parameters.

```
$time_zone (
      )
```

```
Example: "Assume the following is done in Minnesota."
         display_value $date(month)//', '//$time(ampm)
         June 26, 1986, 2:33 PM
         display_value $time_zone
         -6:0.DAYLIGHT_SAVING_TIME
         display_value $time(ampm, $now-$time_zone)
         7:33 PM
```

9.7.9 $time_zone_identifier ($time_zone_id)

The $TIME_ZONE_IDENTIFIER function returns identifing string defined for a time zone. See the section on Time Zone Identifier Modules for information on how to define these strings. If no identifier has been defined for the specified time zone, a null (empty) string is returned.

```
$time_zone_identifier, $time_zone_id (
```

```
      format: any of
          key
            (full, tz, f)
            (brief, tza, b)
          keyend
          string
        anyend = full
      time_zone: time_zone = $time_zone
      )
```

FORMAT: This parameter specifies the format of the returned string as a “time_zone form string,” described in the subsection “Date Time Formats” in the section on “Date Time Expressions.” In addition, a keyword value may be specified for one of the time zone formats; these keywords and their equivalent “time zone form strings” are shown in the following table.

Keyword	Time Zone Form
FULL, TZ, F	'TZ'
BRIEF, TZA, B	'TZA'

Omission causes the full identifier for the time zone in the current natural language to be used.

TIME_ZONE: This parameter specifies the time_zone whose identiifer is to be returned.

Omission causes the system's time zone ($TIME_ZONE) to be used.

```
Example: "Assume the following is done in Minnesota."
         display_value $date(month)//', '//$time(ampm)//..
         ' '//$time_zone_identifier(brief)
         June 26, 1986, 2:33 PM CDT
         display_value ($time_zone, $time_zone_identifier)
         -6:0.DAYLIGHT_SAVING_TIME
         Central Daylight Time
```

9.7.10 $universal_date_time

The $UNIVERSAL_DATE_TIME function returns the “universal” date and time equivalent to a date and time expressed relative to a “local” time zone. The “universal” time zone is that of Greenwich, England, often called Greenwich Mean Time.

```
$universal_date_time (
      local_date_time: date_time = $required
      time_zone: time_zone = $required
      )
```

LOCAL_DATE_TIME: This parameter specifies the “local” date_time value to be converted.

TIME_ZONE: This parameter specifies the “local” time_zone.

9.8 Functions Related to Data Names

9.8.1 $data_name

The $DATA_NAME function interprets the contents of a string as a data_name or converts a name to a data_name.

```
$data_name (
      source: any of
          string
          name
        anyend = $required
```

```
      )
```

SOURCE: This parameter specifies the value to be converted.

9.9 Functions Related to Expressions

9.9.1 $evaluate

The $EVALUATE function is used to evaluate an expression of a specified type or to check whether the evaluation would work.

```
$evaluate (
      expression: any of
          string
          application
        anyend = $required
      type_specification: type
      option: key
          value
          check
        keyend = value
      )
```

EXPRESSION: This parameter specifies the expression to be evaluated.

TYPE_SPECIFICATION: This parameter specifies the data type for the expression.

Omission causes the type the function is expected to return to be used.

OPTION: This parameter specifies whether the result of evaluating the expression is to be returned (VALUE), or a boolean indicating whether the evaluation was successful (CHECK).

Omission causes VALUE to be used.

9.10 Functions Related to Files, Catalogs and Paths

There are more functions than those described below that are useful in dealing with files. All such functions supplied by NOS/VE, including those described here are fully discussed in the File Management section of this document.

The functions described below are included here becauses of their close relationship to file expressions and the programming capabilities of SCL.

9.10.1 $catalog_contents

The $catalog_contents function returns a list of the names of files and/or catalogs contained directly within a catalog. The list is sorted in ascending order.

```
$catalog_contents (
      catalog: file = $working_catalog
      options: list rest of key
          (include_catalogs, include_catalog, ic)
          (include_files, include_file, if)
          (names, name, n)
          (paths, path, p)
        keyend = include_catalogs include_files names
      )
```

CATALOG: This parameter specifies the catalog whose contents are to be returned.

OPTIONS: This parameter controls the form and content of the result list.

INCLUDE_CATALOGS, INCLUDE_CATALOG, IC: This option causes subcatalogs to be included in the result.

INCLUDE_FILES, INCLUDE_FILE, IF: This option causes files to be included in the result.

NAMES, NAME, N: This option causes the elements of the result list to be the names of the catalogs and/or files.

PATHS, PATH, P: This option causes the elements of the result list to be the absolute paths of the catalogs and/or files.

If neither the NAMES or PATHS option is selected, NAMES is assumed (only one of these options may be selected). Omission causes both catalog and files to be included and returned as names.

9.10.2 $family

The $family function returns the absolute path for the family of the current user. It has no parameters.

```
$family (
      )
```

9.10.3 $file_attributes

The $FILE_ATTRIBUTES function return information about files and catalogs. See the File Management ERS for a complete description of the $FILE_ATTRIBUTES function.

9.10.4 $file_cycles

The $FILE_CYCLES function returns a list of the numbers of the cycles of a file or a list of the absolute paths for the cycles of a file. In either case the list is sorted in decending order by cycle number.

```
$file_cycles (
      file: file = $required
      option: key
          (cycles, cycle, c)
          (paths, path, p)
        keyend = cycles
      )
```

If the cycles option is selected, the elements of the result list are the cycle numbers of the file. If the paths option is selected, the elements of the result list are the absolute paths of the file's cycles. Omission of the option parameter casues cycles to be assumed.

FILE: This parameter specifies the file whose cycles are to be returned.

OPTIONS: This parameter controls the form of the result list.

CYCLES, CYCLE, C: This option causes the elements of the result list to be the numbers of the cycles of the file.

PATHS, PATH, P: This option causes the elements of the result list to be the absolute paths of the cycles of the file..

Omission causes the cycle numbers to be returned.

9.10.5 $fname

The $FNAME function interprets the contents of a string or a list as a file reference.

```
$fname (
      source: any of
          string
```

```
        list of any of
          key
            none, $low, $high, $next, $asis, $boi, $bop, $eoi
          keyend
          integer
          name
        anyend
      anyend = $required
    )
```

SOURCE: This parameter specifies either a string or a list to be interpreted as a file expression. If a list is given it consists of one or more name elements optionally followed by a keyword or integer specifying a cycle reference, optionally followed by a keyword designating an open position. If an element contains the keyword NONE, it is ignored.

9.10.6 $high

The $HIGH “function” returns the cycle number of the highest cycle for a catalogued file.

The “function” must be used as a cycle reference function (see the syntax definition for file expressions). For example,

some_file//$high

evaluates to the absolut file references for the highest cycle of “some_file.” This path will designate the cycle number as opposed to the generic cycle reference, $HIGH.

9.10.7 $local

The $LOCAL function returns the absolute path for the catalog which contains the files and subcatalogs that are local to the current job. It has no parameters.

```
$local (
      )
```

9.10.8 $low

The $LOW “function” returns the cycle number of the lowest cycle for a catalogued file.

The “function” must be used as a cycle reference function (see the syntax definition for file expressions). For example,

some_file//$low

evaluates to the absolut file references for the lowest cycle of “some_file.” This path will designate the cycle number as opposed to the generic cycle reference, $LOW.

9.10.9 $next

The $NEXT “function” returns the cycle number of the next highest ($HIGH+1) cycle for a file. If $NEXT is used in conjunction with a file that is not catalogued, the resulting cycle number is 1.

The “function” must be used as a cycle reference function (see the syntax definition for file expressions). For example,

some_file//$next

evaluates to the absolut file references for the next highest cycle of “some_file.” This path will designate the cycle number as opposed to the generic cycle reference, $NEXT.

9.10.10 $path_elements

The $path_elements function is passed a file and returns a list of the elements of the file refernce.

```
$path_elements (
      file: file = $required
      options: list rest of key
          (names, name, n)
          (cycle_reference, cr)
          (open_position, op)
        keyend = names
      )
```

FILE: This parameter specifies the path whose elements are to be returned.

OPTIONS: This parameter specifies which components of the file reference are to be included in the result list.

NAMES, NAME, N: causes all of the path element names to be included as the first of the list.

CYCLE_REFERENCE, CR: causes an element representing a cycle reference to be included. If the file reference contains a specific cycle number, the corresponding integer is returned. If it contains a generic cycle reference, the corresponding keyword is returned. If it does not contain a cycle reference, the keyword NONE is returned.

OPEN_POSITION, OP: causes a keyword corresponding to the open position in the file reference to be returned as the last element of the list. If the file reference does not contain an open position, the keyword NONE is returned.

Omission causes NAMES to be selected.

9.10.11 $procedure (*)

The $PROCEDURE function returns the absolute path for the catalog which contains the files and subcatalogs that are local to the currently executing SCL procedure. It has no parameters. If the function is not called from within a procedure it is equivalent to the $LOCAL function.

```
$procedure (
      )
```

9.10.12 $system

The $SYSTEM function returns the absolute path for the system catalog (i.e. :$SYSTEM.$SYSTEM).

```
$system (
      )
```

9.10.13 $task (*)

The $TASK function returns the absolute path for the catalog which contains the files and subcatalogs that are local to the currently executing task. It has no parameters. If the function is called from within the original task of a job it is equivalent to the $LOCAL function.

```
$task (
      )
```

9.10.14 $up

The $UP function returns the absolute path for the catalog that is one level “up” in the hierarchy from the path it is passed as a parameter.

```
$up (
      path: file = $working_catalog
      )
```

PATH: This parameter specifies the path whose containing catalog is desired.

Omission causes the working catalog to be used.

9.10.15 $user

The $USER function returns the absolute path for the master catalog of the current user. It has no parameters.

```
$user (
      )
```

9.10.16 $working_catalog ($wc)

The $WORKING_CATALOG function returns the path for the working catalog. It has no parameters.

```
$working_catalog, $wc (
      )
```

9.11 Functions Related to Integers

9.11.1 $integer

The $INTEGER returns the integer equivalent of its parameter.

```
$integer (
      source: any of
          string
          real
          boolean
        anyend = $required
      default_radix: integer 2..16 = 10
      )
```

SOURCE: This parameter specifies the value to be converted.

If a string is given, it must contain a signed or unsigned integer constant with optional radix.

If a real number is given, the integer portion of the real number is returned. In effect, the fractional part of the real number is truncated; no rounding occurs.

If a boolean is given, zero is returned for FALSE, and one is returned for TRUE.

DEFAULT_RADIX: This parameter can be used when SOURCE is a string to specify what radix to use if the radix specification is omitted.

Omission causes a decimal radix (10) to be used.

9.11.2 $integer_string

The $INTEGER_STRING function returns the string representation of an integer.

```
$integer_string (
      integer: integer = $required
      radix: integer 2..16 = 10
      include_radix: boolean = no
      )
```

INTEGER: This parameter specifies the integer value.

RADIX: This parameter may be used to specify the radix in which the integer is to be represented.

Omission causes a decimal radix (10) to be used.

INCLUDE_RADIX: This parameter specifies whether to include a standard SCL radix specification in the result (YES) or not (NO).

Omission causes NO to be used.

9.11.3 $max_integer

The $MAX_INTEGER function returns the largest positive integer value. It has no parameters.

```
$max_integer (
      )
```

9.11.4 $min_integer

The $MIN_INTEGER function returns the largest negative integer value. It has no parameters.

```
$min_integer (
      )
```

9.11.5 $mod

The $MOD function returns the modulus on one integer with respect to another. It is computed using to the following formula (the division used is “integer division.”

a - ((a / b) * b)

```
$mod (
      a: integer = $required
      b: integer = $required
      )
```

A: This parameter specifies operand “a” in the above formula.

B: This parameter specifies operand “b” in the above formula.

9.12 Functions Related to Lists

9.12.1 $add

The $ADD function returns a list formed by adding a new element to the front of an existing list.

```
$add (
      element: any = $required
      list: list 0..$max_list = $required
      )
```

ELEMENT: This parameter specifies the value to become the first element of the resulting list.

LIST: This parameter specifies the list to which ELEMENT is added.

9.12.2 $apply

The $APPLY function returns a list formed by applying an expression to each element of another list.

```
$apply (
      list: list 0..$max_list = $required
      expression: (defer) any = $required
      )
```

LIST: This parameter specifies the list whose elements are to be operated upon.

EXPRESSION: This parameter specifies the expression used to compute the elements of the result list.

The EXPRESSION references an element of LIST via the variable, X, which is local to the function. The following example illustrates how $APPLY works.

```
var
  a: list of integer = (9, 6, 42, 369)
  b: list of string
varend

” Given the above variables, the following statement:

b = $apply(a, $justify($integer_string(x), 4, right, '0'))

” is equivalent to the following statements:

b = ()
for each e in a do
  b = $join(b, $justify($integer_string(x), 4, right, '0'))
forend

” and produces

display_value b do=source

('0009', '0006', '0042', '0369')
```

Interpretation of the EXPRESSION is affected by the context in which $APPLY is called. The function determines the data type it is expected to return. If that type is LIST, its element type is used in the evaluation of the EXPRESSION, otherwise type ANY is used. For example,

```
var
  names: list of name = (Carol Donna Robyn)
  id_list: list of record
    id: name
    code: integer
  recend
varend

id_list = $apply(names, (x, 123))
display_value id_list display_option=data_structure
"LIST"
  1: "RECORD"
       ID: "NAME"  CAROL
       CODE: "INTEGER"  123
     "RECORD END"
  2: "RECORD"
       ID: "NAME"  DONNA
       CODE: "INTEGER"  123
```

```
        "RECORD END"
    3: "RECORD"
         ID: "NAME"  ROBYN
         CODE: "INTEGER"  123
        "RECORD END"
  "LIST END"

  display_value $apply(names, (x, 123)) do=data_structure
  "LIST"
    1: "LIST"
         1: "NAME"  CAROL
         2: "INTEGER"  123
        "LIST END"
    2: "LIST"
         1: "NAME"  DONNA
         2: "INTEGER"  123
        "LIST END"
    3: "LIST"
         1: "NAME"  ROBYN
         2: "INTEGER"  123
        "LIST END"
  "LIST END"
```

The difference between the output of the two displays illustrates how $APPLY uses its expected result type to interpret the EXPRESSION for each LIST element.

9.12.3 $build_list

The $BUILD_LIST function returns a list formed by individually computing a specified number of its elements according to a specified expression.

```
$build_list (
      number_of_new_elements: integer 0..$max_list = $required
      next_element: (defer) any = $required
      initial_list: list 0..$max_list = ()
      )
```

NUMBER_OF_NEW_ELEMENTS: This parameter specifies the number of list elments to be generated.

NEXT_ELEMENT: This parameter specifies the expression used to compute the new elements of the result list.

INITIAL_LIST: This parameter specifies a list to which the new elements are appended.

Omission causes an empty list to be used.

The NEXT_ELEMENT expression can reference a record variable, X, which is local to the function. X.RESULT refers to the list built so far and is initially set to the INITIAL_LIST parameter. X.INDEX refers to the index of the element being computed.

```
Example: display_value $build_list(10, x.index) do=source
         (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

         display_value $build_list(8, ..
               x.result(x.index-2)+x.result(x.index-1), ..
               (1, 1)) do=source
         (1, 1, 2, 3, 5, 8, 13, 21, 33, 54)

         s = 'abcdefghij'
         display_value $build_list($size(s), s(x.index)) do=source
         ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
```

9.12.4 $build_result

The $BUILD_RESULT function returns a single result by performing a specified operation on all the elements of a list.

```
$build_result (
      list: list = $required
      expression: (defer) any = $required
      )
```

LIST: This parameter specifies the list.

EXPRESSION: This parameter specifies the expression used to update the result for each successive elements of the LIST.

The EXPRESSION can reference a record variable, X, which is local to the function. X.RESULT refers to the result as computed so far and is initially set to the the first element of LIST. X.CURRENT refers to the element of LIST currently being dealt with. X.INDEX refers to the current element's index within the LIST. (Note that X.CURRENT and X.INDEX will never refer to the first element of the LIST.)

The $BUILD_RESULT function is a generic version of function such as $SUM and $MAX, the former returns the sum of a list of numbers, the latter returns the largest of a list of numbers. As an illustration, given that NUMBERS designates a list of integers and/or reals, the following two calls are equivalent:

```
$sum(numbers)
$build_result(numbers, x.result+x.current)
```

and the following are equivalent:

```
$max(numbers)
$build_result(numbers, $if(x.current>x.result, x.current, x.result))
```

9.12.5 $combine

The $COMBINE function returns a list formed by applying an expression to corresponding elements of other lists. The $COMBINE function can be thought of as a generalized form of the $APPLY function.

```
$combine (
      expression: (defer) any = $required
      lists: list rest of list 0..$max_list = $required
      )
```

EXPRESSION: This parameter specifies the expression used to compute the elements of the result list.

LISTS: This parameter specifies the lists whose corresponding elements are to be operated upon.

The EXPRESSION references the elements of the LISTS via the array variable, X, which is local to the function. X(1) is used to refer to the element from the first list, X(2) refers to the element from the second list, and so on. X(0) refers to the “index” of the elements within their lists. The number of elements in the result list is the number of elements in the shortest input list. The following example illustrates how $COMBINE works.

```
var
  a: list of string = ('A', 'B', 'C', 'D')
  b: list of string = ('x', 'y', 'z')
  c: list of string
varend

” Given the above variables, the following statement:

c = $combine(x(1)//x(0)//x(2), a, b)
```

```
" is equivalent to the following statements:

c = ()
for i = 1 to $size(a) do
  exit when i > $size(b)
  c = $join(c, a(i)//i//b(i))
forend

" and produces

display_value c do=source

('A1x', 'B2y', 'C3z')
```

Interpretation of the EXPRESSION is affected by the context in which $COMBINE is called. The function determines the data type it is expected to return. If that type is LIST, its element type is used in the evaluation of the EXPRESSION, otherwise type ANY is used. For example,

```
var
  names: list of name = (Carol Donna Robyn)
  codes: list of integer = (2, 3, 1)
  id_list: list of record
    id: name
    code: integer
  recend
varend

id_list = $combine((x(1), x(2)), names, codes)
display_value id_list display_option=data_structure
"LIST"
  1: "RECORD"
       ID: "NAME"  CAROL
       CODE: "INTEGER"  2
     "RECORD END"
  2: "RECORD"
       ID: "NAME"  DONNA
       CODE: "INTEGER"  3
     "RECORD END"
  3: "RECORD"
       ID: "NAME"  ROBYN
       CODE: "INTEGER"  1
     "RECORD END"
"LIST END"

display_value $combine((x(1), x(2)), names, codes) ..
      do=data_structure
"LIST"
  1: "LIST"
       1: "NAME"  CAROL
       2: "INTEGER"  2
     "LIST END"
  2: "LIST"
       1: "NAME"  DONNA
       2: "INTEGER"  3
     "LIST END"
  3: "LIST"
       1: "NAME"  ROBYN
       2: "INTEGER"  1
     "LIST END"
"LIST END"
```

The difference between the output of the two displays illustrates how $COMBINE uses its expected result type to interpret the expression.

9.12.6 $difference

The $DIFFERENCE function is passed two lists and returns a list that contains those elements that appear in the first list and do not appear in the second.

The element type of the lists must be the same and must be able to be compared for equality.

```
$difference (
      first: list 0..$max_list = $required
      second: list 0..$max_list = $required
      )
```

FIRST: This parameter specifies the list of elements to be included in the result except for those elements that appear in the SECOND list.

SECOND: This parameter specifies the elements that should not be included in the result.

9.12.7 $first

The $FIRST function is passed a list and returns the value of the first element of the list.

```
$first (
      list: list = $required
      )
```

LIST: This paramter specifies the list whose first element is to be returned.

9.12.8 $intersection

The $INTERSECTION function is passed one or more lists and returns a list containing those elements that appear in each of them.

The element type of the lists must be the same and must be able to be compared for equality.

```
$intersection (
      lists: list rest of list 0..$max_list = $required
      )
```

LISTS: This parameter specifies the lists whose intersection is to be returned.

9.12.9 $join

The $JOIN function returns a list formed by “concatenating” the parameters it is passed. All of the elements of each parameter become elements of the returned list.

```
$join (
      lists: list rest of list 0..$max_list = $required
      )
```

LISTS: This parameter specifies the lists to be joined.

9.12.10 $last

The $last function is passed a list and returns the value of the last element of the list.

```
$last (
      list: list = $required
      )
```

LIST: This paramter specifies the list whose last element is to be returned.

9.12.11 $list_of

The $LIST_OF function returns a list containing the parameters it is passed. As contrasted to the $JOIN function, the $LIST_OF function does not make each element of a list parameter an element of the returned list; rather each parameter becomes an individual element of the returned list.

```
$list_of (
      elements: list rest of any = $required
      )
```

ELEMENTS: This paramter specifies the elements of the list to be returned.

9.12.12 $max

The $MAX function returns the maximum number within a list of numbers (integers and/or reals).

```
$max (
      numbers: list rest of any of
          integer
          real
        anyend = $required
      )
```

NUMBERS: This parameter specifies the list of numebrs.

9.12.13 $max_list

The $MAX_LIST function returns an integer representing the maximum number of elements that a list value may have. It has no parameters.

```
$max_list (
      )
```

9.12.14 $min

The $MIN function returns the maximum number within a list of numbers (integers and/or reals).

```
$min (
      numbers: list rest of any of
          integer
          real
        anyend = $required
      )
```

NUMBERS: This parameter specifies the list of numebrs.

9.12.15 $nil

The $NIL function is passed a list and returns a boolean value indicating whether the list is empty (TRUE if empty and FALSE otherwise). The expression “$NIL(x)” is equivalent to the expression “$SIZE(x)=0.”

```
$nil (
      list: list 0..$max_list = $required
      )
```

LIST: This parameter specifies the list to be tested.

9.12.16 $rest

The $REST function is passed a list and returns a list consisting of all of the elements of that list except the first.

```
$rest (
      list: list = $required
      )
```

LIST: This paramter specifies the list whose elements, all but the first, are to be returned.

9.12.17 $reverse

The $REVERSE function returns a list formed by reversing the order of the elements in the list passed to it.

```
$reverse (
      list: list 0..$max_list = $required
      )
```

LIST: This paramter specifies the list whose elements are to be returned in reverse order.

9.12.18 $select

The $SELECT function is passed a list and a boolean expression and returns a list containing each element of the original list for which the boolean expression evaluates to true.

If the expected result type for the function in the context in which it is called is “list of integer,” a list of the indices of the matching elements is returned instead of a list of the elements themselves.

```
$select (
      list: list 0..$max_list = $required
      condition: (defer) boolean = $required
      )
```

LIST: This parameter specifies the list from whose elements the resulting elements are to be selected.

CONDITION: This parameter specifies the boolean expression used to determine whether an element of LIST should be included in the result list.

Within the CONDITION expression, the element of LIST can be referenced via the variable X, which is local to the function. For example

$select(strings, x(4,1)='$')

selects those elements from the list “strings” whose fourth character is a '$'.

9.12.19 $select_string(s)

The $SELECT_STRINGS function provides the ability to use string patterns with a list of strings. It returns a list of those candidate strings that match the specified pattern. If no matches are found, an empty list is returned.

If the expected result type for the function in the context in which it is called is “list of integer,” a list of the indices of the matching elements is returned instead of a list of the elements themselves.

```
$select_string, $select_string (
      candidates: list 0..$max_list of string = $required
      pattern: string_pattern = $required
      anchor_option: key
          (anchored, a)
          (unanchored, u)
        keyend = unanchored
      scan_option: (advanced) key
          (quick_scan, qs)
          (full_scan, fs)
        keyend = quick_scan
      )
```

CANDIDATES: This parameter specifies a list of program names from which are selected those that match the PATTERN.

PATTERN: This parameter specifies the pattern. String patterns are described in the sections on *String Pattern Expressions* and *Functions Releated to String Patterns*.

ANCHOR_OPTION: For an explanation of this parameter, see the $MATCH string pattern function.

SCAN_OPTION: For an explanation of this parameter, see the $MATCH string pattern function.

9.12.20 $select_wild_card_file(s), $select_file(s)

The $SELECT_WILD_CARD_FILES function provides the ability to use “wild cards” with an existing list of files. It returns a list of those candidate files that match the specified pattern. If no matches are found, an empty list is returned.

If the expected result type for the function in the context in which it is called is “list of integer,” a list of the indices of the matching elements is returned instead of a list of the elements themselves.

```
$select_wild_card_files, $select_wild_card_file, $select_file, $select_files (
      candidates: list 0..$max_list of file = $required
      pattern: (wild_card_file) application = $required
      )
```

CANDIDATES: This parameter specifies a list of files from which are selected those that match the PATTERN.

PATTERN: This parameter specifies the pattern which consists of the “wild card” notation described in the sections on *File Expressions* and “*Wild Card” Patterns.*

9.12.21 $select_wild_card_name(s), $select_name(s)

The $SELECT_WILD_CARD_NAMES function provides the ability to use “wild cards” with a list of names. It returns a list of those candidate names that match the specified pattern. If no matches are found, an empty list is returned.

If the expected result type for the function in the context in which it is called is “list of integer,” a list of the indices of the matching elements is returned instead of a list of the elements themselves.

```
$select_wild_card_names, $select_wild_card_name, $select_name, $select_names (
      candidates: list 0..$max_list of name = $required
```

```
      pattern: (wild_card_name) application = $required
      pattern_type: key
          (basic, b)
          (extended, e)
        keyend = $scl_options.wild_card_pattern_type
      )
```

CANDIDATES: This parameter specifies a list of names from which are selected those that match the PATTERN.

PATTERN: This parameter specifies the pattern which consists of the "wild card" notation described in the section on "*Wild Card" Patterns.*

PATTERN_TYPE: This parameter specifies whether the PATTERN is to be interpreted as a BASIC or EXTENDED "wild card" pattern. See the section on "*Wild Card" Patterns* for more infomration.

Omission causes the value of the SCL option WILD_CARD_PATTERN_TYPE to be used.

9.12.22 $select_wild_card_program_name(s)

The `$SELECT_WILD_CARD_PROGRAM_NAMES` function provides the ability to use "wild cards" with a list of program names. It returns a list of those candidate program names that match the specified pattern. If no matches are found, an empty list is returned.

If the expected result type for the function in the context in which it is called is "list of integer," a list of the indices of the matching elements is returned instead of a list of the elements themselves.

```
$select_wild_card_program_names, $select_wild_card_program_name (
      candidates: list 0..$max_list of program_name = $required
      pattern: (wild_card_name) application = $required
      pattern_type: key
          (basic, b)
          (extended, e)
        keyend = $scl_options.wild_card_pattern_type
      )
```

CANDIDATES: This parameter specifies a list of program names from which are selected those that match the PATTERN.

PATTERN: This parameter specifies the pattern which consists of the "wild card" notation described in the section on "*Wild Card" Patterns.*

PATTERN_TYPE: This parameter specifies whether the PATTERN is to be interpreted as a BASIC or EXTENDED "wild card" pattern. See the section on "*Wild Card" Patterns* for more infomration.

Omission causes the value of the SCL option WILD_CARD_PATTERN_TYPE to be used.

9.12.23 $select_wild_card_string(s)

The `$SELECT_WILD_CARD_STRINGS` function provides the ability to use "wild cards" with a list of strings. It returns a list of those candidate strings that matched the specified pattern. If no matches are found, an empty list is returned.

If the expected result type for the function in the context in which it is called is "list of integer," a list of the indices of the matching elements is returned instead of a list of the elements themselves.

(See the $SELECT_STRINGS function if you need to use a more complex pattern than can be represented using the "wild card" notation.)

```
$select_wild_card_string, $select_wild_card_string (
      candidates: list 0..$max_list of string = $required
      pattern: (wild_card_pattern) any of
          string
          application
        anyend = $required
      pattern_type: key
```

```
          (basic, b)
          (extended, e)
        keyend = $scl_options.wild_card_pattern_type
      )
```

CANDIDATES: This parameter specifies a list of program names from which are selected those that match the PATTERN.

PATTERN: This parameter specifies the pattern which consists of the "wild card" notation described in the section on "*Wild Card" Patterns.* A string can be used to specify the pattern if it contains SCL delimiter characters.

PATTERN_TYPE: This parameter specifies whether the PATTERN is to be interpreted as a BASIC or EXTENDED "wild card" pattern. See the section on "*Wild Card" Patterns* for more infomration.

Omission causes the value of the SCL option WILD_CARD_PATTERN_TYPE to be used.

9.12.24 $size

The $SIZE function is passed a list and returns the number of elements in the list.

```
$size (
      list: list 0..$max_list = $required
      )
```

LIST: This paramter specifies the list whose number of elements is to be returned.

9.12.25 $sort

The $SORT function is passed a list and returns a list containing the same elements sorted in an order determined by an expression also given to the function.

```
$sort (
      list: list 0..$max_list = $required
      order: (defer) any of
          key
            (ascending, a)
            (descending, d)
          keyend
          boolean
        anyend = ascending
      )
```

LIST: This paramter specifies the list whose elements are to be returned in sorted order.

ORDER: This parameter determines the order of the elements in the resulting list. If the keyword ASCENDING (A) is specified the list is sorted in ascending order treating the entire element as the "sort key." If the keyword DESCENDING (D) is specified the list is sorted in descending order treating the entire element as the "sort key."

Alternatively, a boolean expression may be used to specify the resulting order. The function uses this expression to repeatedly ask the question: "should element X1 come before element X2." The variables "x1" and "x2" in the "order" expression are local to the function. For example, the ordering expression "x1<=x2" is equivalent to the ASCENDING keyword and "x1>=x2" is equivalent to the DESCENDING keyword.

Omission causes ASCENDING to be used.

9.12.26 $sublist

The $SUBLIST function returns the elements of a list selected according to their indices.

```
$sublist (
```

```
      list: list 0..$max_list = $required
      selector: list 0..$max_list of any of
          integer 1..$max_list
          range of integer 1..$max_list
          record
            start: integer 1..$max_list
            count: any of
              key
                all
              keyend
              integer 0..$max_list
            anyend
          recend
        anyend = $required
      )
```

LIST: This parameter specifies the list from which the elements are selected.

SELECTOR: This parameter specifies the indices of the elements that should be included in the result.

```
Example: a_list = (a, b, c, d, e, f, g, h, i, j)

         display_value $sublist(a_list, (2, 5, 8)) do=source
         (B, E, H)

         display_value $sublist(a_list, ((4, all))) do=source
         (D, E, F, G, H, I, J)

         display_value $sublist(a_list, 3..5) do=source
         (C, D, E)



display_value $sublist(a_list, (2, 5..8, 1, 5)) do=source
         (B, E, F, G, H, A, E)
```

9.12.27 $subset

The $SUBSET function is passed two lists and returns a boolean indicating whether all of the elements of the first list are also elements of the second list (i.e., is the first list a subset of the second list).

The element type of the two lists must be the same and must be able to be compared for equality.

```
$subset (
      subset: list 0..$max_list = $required
      set: list 0..$max_list = $required
      )
```

SUBSET: This parameter specifies the list of elements to be looked for in the SECOND list.

SET: This parameter specifies the full set of elements.

9.12.28 $sum

The $SUM function returns the result of adding a list of numbers (integers and/or reals) together. If the list is empty, zero is returned.

```
$sum (
      numbers: list rest 0..$max_list of any of
```

```
          integer
          real
        anyend = $required
      )
```

NUMBERS: This parameter specifies the list of numebrs to be summed.

9.12.29 $union

The $UNION function is passed one or more lists and returns a list containing the elements of all of them. It differs from the $join function in that $union eliminates duplicate occurrences of the elements in the list it returns.

The element type of the lists must be the same and must be able to be compared for equality.

```
$union (
      lists: list rest of list 0..$max_list = $required
      )
```

LISTS: This parameter specifies the lists whose union is to be returned.

9.12.30 $wild_card_file(s) ($wcf)

This functions enables a user to control whether only files, only catalogs, or both are included when a file reference containing “wild cards” is expanded. (When the expression evaluator performs such an expansion, the result includes only files.) If no matches are found, an empty list is returned.

```
$wild_card_files, $wild_card_file, $wcf (
      files: (wild_card_file) application = $required
      options: list rest of key
          (include_catalogs, ic)
          (include_files, if)
        keyend = include_catalogs include_files
      )
```

FILES: This parameter specifies a file reference containing the “wild card” notation discussed in the sections on *File Expressions* and *“Wild Card” Patterns*.

OPTIONS: This parameter specifies whether files, catalogs or both are to be included in the result list.

Omission causes both files and catalogs to be included.

9.13 Functions Related to Locks

9.13.1 $locked (*)

The $LOCKED function returns a boolean value of TRUE if its lock parameter is set and FALSE otherwise. A lock is set by the LOCK statement and cleared by the UNLOCK statement.

```
$locked (
      lock: lock = $required
      )
```

LOCK: This parameter specifies the lock to be tested.

9.14 Functions Related to Names

9.14.1 $max_name

The $MAX_NAME function returns an integer representing the maximum number of characters that can be used in a name. It has no parameters.

```
$max_name (
      )
```

9.14.2 $name

The $NAME function interprets the contents of a string as a name.

```
$name (
      string: string = $required
      )
```

STRING: This parameter specifies the string containing an SCL name.

9.15 Functions Related to Parameters

9.15.1 $specified

The $SPECIFIED function returns an answer to the question: “was this parameter supplied on the call to the command or function procedure?”

This function can be used to distinguish between a default value and one actually supplied on a command or function call.

```
$specified (
      parameter: data_name = $required
      )
```

PARAMETER: This parameter specifies the name of the procedure parameter to be checked.

9.15.2 $parameter_value

The $PARAMETER_VALUE function returns the value of a parameter.

This function provides a way to obtain the value of a procedure parameter when a variable has been created within the procedure with the same name as the parameter. A reference to the name, in such a context would designate the variable.

```
$parameter_value (
      parameter: data_name = $required
      )
```

PARAMETER: This parameter specifies the name of the procedure parameter whose value is to be returned.

9.16 Functions Related to Program Names

9.16.1 $program_name

The $PROGRAM_NAME function interprets the contents of a string as a program_name or converts a COBOL_name or name to a program_name.

```
$program_name (
      source: any of
          string
          cobol_name
          name
        anyend = $required
      )
```

SOURCE: This parameter specifies the value to be converted.

9.17 Functions Related to Ranges

9.17.1 $range_of

The $RANGE_OF function returns a range value.

```
$range_of (
      low: any = $required
      high: any
      )
```

LOW: This parameter specifies the LOW value for the range.

HIGH: This parameter specifies the HIGH value for the range.

Omission causes the low and high range elements to designate the same value.

For example,

```
$range_of(a, b)
```

is equivalent to

```
a..b
```

and

```
$range_of(a)
```

is equivalent to

```
a
```

supplied for a range expression.

9.17.2 $range_specified

The $RANGE_SPECIFIED function answers the question “was this range value” specified as a range?,” i.e. were both the low and high components given when the range value was constructed.

```
$range_specified (
      range: range = $required
```

```
      )
```

RANGE: This parameter specifies the range value to be checked.

```
Example: display_value $range_specified(123..456)
         TRUE
         display_value $range_specified(123)
         FALSE
         display_value $range_specified(123..123)
         TRUE
```

9.18 Functions Related to Real Numbers

9.18.1 $indefinite

The $INDEFINITE function returns a boolean value of TRUE if its real number parameter is “indefinite” and FALSE otherwise.

```
$indefinite (
      real_number: real = $required
      )
```

REAL_NUMBER: This parameter specifies the real value to be checked.

9.18.2 $infinite

The $INFINITE function returns a boolean value of TRUE if its real number parameter is “infinite” and FALSE otherwise.

```
$infinite (
      real_number: real = $required
      )
```

REAL_NUMBER: This parameter specifies the real value to be checked.

9.18.3 $infinity

The $INFINITY function returns a real number that represents “infinity.” It has no parameters.

```
$infinity (
      )
```

9.18.4 $max_real

The $MAX_REAL function returns the largest positive real number that can be represented as less than positive “infinity.” It has no parameters.

```
$max_real (
      )
```

9.18.5 $min_real

The $MIN_REAL function returns the largest negative real number that can be represented as greater than negative "infinity." It has no parameters.

```
$min_real (
      )
```

9.18.6 $real

The $REAL function returns the real number equivalent of its parameter.

```
$real (
      source: any of
          string
          integer
        anyend = $required
      )
```

SOURCE: This parameter specifies the value to be converted.

If a string is given, it must contain a signed or unsigned real or integer constant.

If an integer is given, it is converted to its corresponding real number representation.

9.18.7 $real_string

The $REAL_STRING function returns the string representation of a real number.

```
$real_string (
      real_number: real = $required
      max_digits: integer 1..28 = 28
      )
```

SOURCE: This parameter specifies the value to be converted.

MAX_DIGITS: This parameter may be used to constrain the number of digits that appear in the string representation. It specifies a maximum since trailing zeros after the decimal point are not included.

Omission causes the maximum number of digits that can be represented in the machines real number format to be used.

9.19 Functions Related to Records

9.19.1 $field

The $FIELD function accesses a field of a record. It is useful in those circumstances when the field being referenced is not known until execution time.

```
$field (
      record: any = $required
      name: name
      option: key
          defined
          initialized
          specified
          value
```

```
        keyend = value
      )
```

RECORD: This parameter specifies the record value to be interrogated.

NAME: This parameter specifies the name of a field of RECORD to be interrogated.

Omission causes a list of the names of the fields of RECORD to be returned. In this case the OPTION parameter is ignored.

OPTION: This parameter specifies the attribute of the NAME field to be returned.

DEFINED: This option causes a boolean value of TRUE to be returned if the NAME is a field of RECORD, and FALSE otherwise.

INITIALIZED: This option causes a boolean value of TRUE to be returned if the NAME field of RECORD has been given a value, and FALSE otherwise.

SPECIFIED: This option causes a boolean value of TRUE to be returned if the NAME field of RECORD has been given a value other than an “unspecified” value, and FALSE otherwise.

VALUE: This option causes the value of the NAME field of RECORD to be returned.

Omission causes VALUE to be used.

9.19.2 $field_list

The $FIELD_LIST function returns a list of the fields in a record.

```
$field_list (
      record: any = $required
      option: key
          (names, name, n)
          (values, value,v)
          (specified_fields_only, sfo)
        keyend = value
      )
```

RECORD: This parameter specifies the record whose fields are to be returned in a list.

OPTION: This parameter specifies the form and content of the returned list.

If both the NAMES and VALUES options are specified the result is a list whose type is:

```
TYPE
  field_list = list of record
    name: name
    value: any = $optional
  recend
TYPEND
```

The NAME field holds the name of the corresponding field from the original RECORD and the VALUE field holds the corresponding value.

If the NAMES option is specified but the VALUES option is not, the result is a list of the names of the fields of the original RECORD.

If the VALUES option is specified but the names option is not, the result is a list of the values of the fields of the original RECORD. In this case the SPECIFIED_FIELDS_ONLY option is automatically selected.

If the SPECIFIED_FIELDS_ONLY option is selected, only those fields of the original RECORD to which values have been assigned are included in the result.

If neither the NAMES nor VALUES options are specified, they are both automatically selected.

```
Example: type
           entry = record
```

```
    symbol: string
    definition: integer = $optional
    references: list of integer
  recend
typend
var
  s1: entry = ('first_symbol',,(10,20))
  s2: entry = ('second_symbol',5,())
varend

display_value $field_list(s1) do=data_structure
"LIST"
  1: "RECORD"
       NAME: "NAME"  SYMBOL
       VALUE: "STRING"  'first_symbol'
     "RECORD END"
  2: "RECORD"
       NAME: "NAME"  DEFINITION
       VALUE: "UNINITIALIZED VALUE"
     "RECORD END"
  3: "RECORD"
       NAME: "NAME"  REFERENCES
       VALUE: "LIST"
                1: "INTEGER"  10
                2: "INTEGER"  20
              "LIST END"
     "RECORD END"
"LIST END"

display_value $field_list(s1, names) do=data_structure
"LIST"
  1: "NAME"  SYMBOL
  2: "NAME"  DEFINITION
  3: "NAME"  REFERENCES
"LIST END"

display_value $field_list(s1, values) do=data_structure
"LIST"
  1: "STRING"  'first_symbol'
  2: "LIST"
       1: "INTEGER"  10
       2: "INTEGER"  20
     "LIST END"
"LIST END"

display_value $field_list(s2, values) do=data_structure
"LIST"
  1: "STRING"  'second_symbol'
  2: "INTEGER"  5
  3: "EMPTY LIST"
"LIST END"

display_value $record($field_list(s1, specified_fields_only)) ..
      do=data_structure
"RECORD"
  SYMBOL: "STRING"  'first_symbol'
  REFERENCES: "LIST"
                1: "INTEGER"  10
                2: "INTEGER"  20
              "LIST END"
```

```
          "RECORD END"
```

9.19.3 $record

The $RECORD function constructs a record value from a list of name/value pairs. It is useful in situations where formally declaring a record's type is awkward.

```
$record (
      fields: list rest of record
          name: name
          value: any = $optional
        recend = $required
      )
```

FIELDS: This parameter specifies the name/value pairs from which the result record is to be constructed. Each name must be unique.

```
Example: " In the following function procedure, the $record function is used
          " to turn a list of name/value pairs into a result record.

          FUNCTION $miscellaneous (
            things: list rest of key
                (date, d)
                (time, t)
                (time_zone, tz)
                (working_catalog, wc)
              keyend = $required
            )

            VAR
              item: record
                  name: name
                  value: any
                recend
              result: list of $type(item) = ()
              thing: $element_type(things)
            VAREND

            FOR EACH thing IN things DO
             item.name = thing
              IF item.name = date THEN
                item.value = $date
              ELSEIF item.name = time THEN
                item.value = $time
              ELSEIF item.name = time_zone THEN
                item.value = $time_zone_identifier
              ELSEIF item.name = working_catalog THEN
                item.value = $working_catalog
              IFEND
              result = $join(result, item)
            FOREND

            EXIT WITH $record(result)

          FUNCEND $miscellaneous

          " Here's a sample call:
```

```
display_value $miscellaneous(t, tz, d) do=labeled_elements

Time      : '13:19:58'
Time_Zone : 'Central Standard Time'
Date      : '1990-01-19'
```

9.19.4 $sort_fields

The $SORT_FIELDS function accepts a record value and returns a new record with the fields of the original record sorted alphabetically by field name.

```
$sort_fields (
      record: any = $required
      )
```

RECORD: This paramter specifies the record whose fields are to be sorted.

9.20 Functions Related to Statistic Codes

9.20.1 $statistic_code

The $STATISTIC_CODE function returns the integer form of the specified statistic code.

```
$statistic_code (
      statistic_code: statistic_code = $required
      )
```

STATISTIC_CODE: This parameter specifies the statistic_code value. See the section of statistic_code expressions for details of the various ways of specifying a statistic code.

9.20.2 $statistic_code_string

The $STATISTIC_CODE_STRING function returns the string form of the specified statistic code.

```
$statistic_code_string (
      statistic_code: statistic_code = $required
      )
```

STATISTIC_CODE: This parameter specifies the statistic_code value. See the section of statistic_code expressions for details of the various ways of specifying a statistic code.

9.21 Functions Related to Statuses and Status Codes

9.21.1 $previous_status

The $PREVIOUS_STATUS function is used to obtain the completion status of the previous command. It has no parameters.

```
$previous_status (
      )
```

9.21.2 $status

The $STATUS function returns a status value constructed from its parameters.

```
$status (
      normal: boolean = $required
      identifier: string 2
      condition: status_code
      text: list rest of any
      )
```

NORMAL: This parameter specifies whether the status is to represent a normal status (TRUE), or “abnormal” (FALSE). If NORMAL is TRUE, the remaining parameters must be omitted. If normal is FALSE, the CONDITION parameter must be given.

IDENTIFIER: This parameter specifies the product identifier portion of the status condition code. Typically this parameter is ignored, but exists for compatibility with previous system versions. It is only used in those cases where the CONDITION parameter specifies a number that is less than or equal to 0ffffff(16), in which case it is combined with that number to form the actual condition code.

CONDITION: This parameter specifies the status's code. See the section of status_code expressions for details of the various ways of specifying a status condition code. If a <status code name> is specified but no definition can be found for it, the code corresponding to OSE$UNDEFINED_CONDITION is used.

TEXT: This parameter specifies the optional message parameters that are edited into the message corresponding to the status. The text field of the status record is initialized to empty, then each text value is converted to a string

(if necessary), prefixed with the ASCII unit separator character (US, $char(31)), and appended to the text field of the status record.

9.21.3 $status_code

The $STATUS_CODE function returns the integer form of the specified status condition code.

```
$status_code (
      status_code: status_code = $required
      )
```

STATUS_CODE: This parameter specifies the status_code value. See the section of status_code expressions for details of the various ways of specifying a status condition code. If a <status code name> is specified but no definition can be found for it, zero is returned.

9.21.4 $status_code_name

The $STATUS_CODE_NAME function returns the name of the specified status condition code.

```
$status_code_name (
      status_code: status_code = $required
      )
```

STATUS_CODE: This parameter specifies the status_code value. See the section of status_code expressions for details of the various ways of specifying a status condition code. If a <status code name> is specified but no definition can be found for it, the name UNKNOWN_CONDITION is returned.

9.21.5 $status_code_string

The $STATUS_CODE_STRING function returns the string form of the specified status condition code.

```
$status_code_string (
      status_code: status_code = $required
      )
```

STATUS_CODE: This parameter specifies the status_code value. See the section of status_code expressions for details of the various ways of specifying a status condition code. If a <status code name> is specified but no definition can be found for it, the string '?? 0' is returned.

9.21.6 $status_message

The $STATUS_MESSAGE function returns the message corresponding to a status as a list of strings.

```
$status_message (
      status: status = $required
      maximum_line_size: integer 32..3961 = $required
      message_level: key
          (current, c)
          (brief, b)
          (full, f)
        keyend = current
      )
```

STATUS: This parameter specifies the status to be formatted into a message.

MAXIMUM_LINE_SIZE: This parameter limits the length of each string in the list of strings that represents message. Normally, this is given as the page width of the file to which the message will eventually be written.

MESSAGE_LEVEL: This parameter determines the amount of information that is included in the message. See the section on message levels for details on the meaning of the various levels. The CURRENT value is the one selected via the CHANGE_MESSAGE_LEVEL command.

Omission causes CURRENT to be used.

9.21.7 $status_severity

The $STATUS_SEFVERITY function is used to interrogate the severity of a status condition.

```
$status_severity (
      status_code: status_code = $required
      level: key
          (non_standard, ns, n)
          (dependent, d)
          (informative i)
          (warning w)
          (error e)
          (fatal f)
          (catastrophic c)
        keyend
      )
```

STATUS_CODE: This parameter specifies the status_code value. See the section of status_code expressions for details of the various ways of specifying a status condition code. If no definition for the specified condition code can be found, a severity level of ERROR is assumed.

LEVEL: If this parameter is given, a boolean value of TRUE is returned if the severity of the specified condition is greater than or equal to the specified level, and FALSE otherwise. The keywords are defined above in increasing order by severity level.

Omission causes the keyword corresponding to the severity of the condition to be returned.

9.22 Functions Related to Strings

9.22.1 $char

The $CHAR function returns one or more characters whose ordinal (position) within the ASCII collating sequence is specified.

```
$char (
      ascii: list rest of any of
          key
            (nul, ^@)
            (soh, ^a)
            (stx, ^b)
            (etx, ^c)
            (eot, ^d)
            (enq, ^e)
            (ack, ^f)
            (bel, ^g)
            (bs,  ^h)
            (ht,  ^i)
            (lf,  ^j)
            (vt,  ^k)
            (ff,  ^l)
            (cr,  ^m)
            (so,  ^n)
            (si,  ^o)
            (dle, ^p)
            (dc1, ^q)
            (dc2, ^r)
            (dc3, ^s)
            (dc4, ^t)
            (nak, ^u)
            (syn, ^v)
            (etb, ^w)
            (can, ^x)
            (em,  ^y)
            (sub, ^z)
            (esc, ^[)
            (fs,  ^\)
            (gs,  ^])
            (rs,  ^^)
            (us,  ^_)
            sp
            del
          keyend
          integer 0..255
          string
        anyend = $required
      )
```

ASCII: This parameter specifies the characters to be returned as a string.

A character may be specified by a keyword specifying a standard ASCII symbol for a characters with ordinals 0 (NUL) through 32 (SP) or 127 (DEL).

Alternatively, a two character keyword whose first character is “^” may be used to specify one of the “control” characters 0 (^@) through 31 (^_).

An integer in the range 0 to 255 may be used to specify a character's ordinal.

Also, for convenience, a string may be used to specify one or more characters. This makes it easier to create result strings that are a combination of control and printable characters.

9.22.2 $format_value

The $FORMAT_VALUE function returns a string or a list of string created using a format and an SCL value.

```
$format_value (
      format_string: string = $required
      values: any = $required
      max_string: integer 3..65535 = $max_string
      )
```

FORMAT_STRING: This parameter specifies the format of the resulting “lines.” This string consists of display characters and formatting directives. The display characters are copied to the text lines directly. The format directives all start with a '+' character and have the following effects. (nn'th item here refers to the nn'th item of a list, array, or record).

+Enn Define a soft end of line. This position will be used to break the line if the line extends past the string width. nn defines the indentation for the new line.

+Fc Define a fill character 'c'. This must be immediately followed by another directive with no interveening '+' character. The fill character is used with the +W, +H, and +X directives.

+Hnn Horizontally tab to column. If nn is specified, it tabs to column nn starting a new line if the current column is too large. If nn is ommitted, it tabs to the next tab of 9, 17, 25, etc. The fill character is used to pad the line to the column.

+K Keep the text between pairs of +K directives together on one line.

+Lxnn Puts the specified item label to the string. If nn is specified and not 0, the nn'th item label is used. If nn is not specified, the next item label is used. If nn is 0, the label of the previously referenced item is used. The 'x' specifies how to show record labels. 'U' specifies that they be in upper case characters. 'L' specifies that they be in lower case characters. 'I' specifies that only the initial letters of words be capitalized. If the 'x' is not specified, 'L' is used.

+Nnn Specifies a hard end of line. This causes the current line to be ended and a new line started with nn leading spaces.

+Pxnn Convert the specified item using element representation. If nn is specified and not 0, the nn'th item is used. If nn is not specified, the next item is used. If nn is 0, the previously referenced item is used. The 'x' specifies how to convert names. 'U' specifies that names be in upper case characters. 'L' specifies that names be in lower case characters. 'I' specifies that only the initial letters of words be capitalized. If the 'x' is not specified, 'L' is used.

+R Causes a portion of the format to be repeated. The portion repeated is between '+R' pairs at the same nesting level (see '+('. The repeated format is processed until all items at this nesting level are used. If all items have been used when the initial '+R' is encountered, the repeated portion is not processed at all.

+Sxnn Convert the specified item using source representation. If nn is specified and not 0, the nn'th item is used. If nn is not specified, the next item is used. If nn is 0, the previously referenced item is used. The 'x' specifies how to convert names. 'U' specifies that names be in upper case characters. 'L' specifies that names be in lower case characters. 'I' specifies that only the initial letters of words be capitalized. If the 'x' is not specified, 'L' is used.

+Wjnn Define a fixed field width for the display of the item that follows as 'nn'. If the item is larger than this width, it will be truncated (on the left for right justification; on the right for left justification and center justification). If the item is smaller than this width, it will be justified as specified and padded by the fill character as defined by the 'F' directive. 'j' optionally specifies the justification. 'R' selects right justification, 'L' selects left justification (the default) and 'C' selects center justification. If nn is not specified a item of 31 is used. NOTE: This must be immediately followed by another directive with no interveening '+' character.

+Xnn Expand count as fill character. The fill character is put to the string nn times.

+(nn Move to the next nesting level. This causes the '+P' and '+S' directives to refer to subitems of the next (or nnth) item.

+) Return to the previous nesting level.

++ Places a '+' into the string.

+- Places the null string '' into the string. This allows one to follow a directive by a number without the interpretation of the number as part of the directive. eg. +p+-5 means put the next parameter to the string then

put '5' to the string.

VALUES: This parameter specifies the value to be formatted.

MAX_STRING: This parameter specifies the maximum length of strings in the representation of the value.

```
Example: display_value $format_value('+p+r, +p+r', $catalog_contents)
         " Displays the names of the files in the catalog as a list
         " seperated by commas.

         display_value $format_value('+r+wp  +p+n+r' $catalog_contents)
         " Displays the names of the files in the catalog in two columns.

         display_value $format_value( ..
               'There are +p2 files in catalog +p1.',  ..
               ($working_catalog $size($catalog_contents($wc if))))
         There are 14 files in catalog :cobalt.brh.genu.
```

9.22.3 $justify

The $JUSTIFY function returns a string “justified” (aligned) within a specified number of character positions. The result may be left justified, right justified or centered.

```
$justify (
      subject: string = $required
      size: integer 0..$max_string = $required
      align: key
          (center, c)
          (left, l)
          (right, r)
        keyend = $required
      fill: string 1 = ' '
      )
```

SUBJECT: This parameter specifies the string to be “justified.”

SIZE: This parameter specifies the number of characters to be included in the result. If SIZE is less than the size of SUBJECT, the result consists of the leftmost SIZE characters of SUBJECT.

ALIGN: This parameter determines on which side SUBJECT is padded to produce the result. Selecting LEFT alignment causes padding on the right. Selecting RIGHT alignment causes padding on the left. Selecting CENTER alignment causes padding to be divided between the left and right.

FILL: This parameter is only used when SIZE is greater than $size(subject) and specifies the character used to pad the resulting string to the requested length.

Omission causes the space character to be used.

9.22.4 $max_string

The $MAX_STRING function returns an integer representing the maximum number of characters that can be used in a string. It has no parameters.

```
$max_string (
      )
```

9.22.5 $ord

The $ORD function returns the ordinal or position of its character parameter in the ASCII collating sequence.

```
$ord (
      char: string 1 = $required
      )
```

CHAR: This parameter specifies the character whose ordinal is to be returned.

9.22.6 $quote

The $QUOTE function is used to “quotate” a string.

```
$quote (
      string: string = $required
      )
```

STRING: This parameter specifies the string yo be “quotated.”

```
Example: var
           s: string
           q: string
         varend

         s = 'ABC''DEF'
         q = $quote(s)
         display_value s
         ABC'DEF
         display_value q
         'ABC''DEF'
```

9.22.7 $scan_any

The $SCAN_ANY function is used to search a string for any one of a set of characters and return the index in the string of the found character. If no character from the set appears in the string, zero is returned.

```
$scan_any (
      characters: string 1..256 = $required
      string: string = $required
      )
```

CHARACTERS: This parameter specifies the characters to be searched for in STRING.

STRING. This parameter specifies the string to be searched.

```
Example: var
           digits: string = '0123456789'
           s: string
         varend

         s = 'TEMP_32'
         display_value $scan_any(digits, s)
         6
```

9.22.8 $scan_not_any

The $scan_not_any function is used to search a string for any character that is not in a set of characters and return the index in the string of the found character. If only characters from the set appear in the string, zero is returned.

```
$scan_not_any (
      characters: string 1..256 = $required
      string: string = $required
      )
```

CHARACTERS: This parameter specifies the characters to be searched for in STRING.

STRING. This parameter specifies the string to be searched.

```
Example: var
           digits: string = '0123456789'
           s: string
         varend

         s = 'TEMP_32'
         display_value $scan_not_any(digits, s)
         1
```

9.22.9 $scan_string

The $scan_string function is used to search a string for a pattern of characters, and return the index of the first character of the pattern in the string. If the pattern is the null string, one is returned. If the pattern is not found in the string, zero is returned.

```
$scan_string (
      pattern: string_pattern = $required
      string: string = $required
      )
```

PATTERN: This parameter specifies the string_pattern to be searched for in STRING.

STRING. This parameter specifies the string to be searched.

```
Example: var
           s: string
           p: string_pattern
         varend

         s = '123_abc_9'
         p = 'abc'
         display_value $scan_string(p, s)
         5
```

9.22.10 $size

The $SIZE function is passed a string and returns the number of characters in the string.

```
$size (
      string: string = $required
      )
```

STRING: This parameer specifies the string whose size is to be returned.

9.22.11 $string

The $STRING function returns the string representation of its first parameter. It returns a single string if the representation

fits in one string and returns a list of strings otherwise.

```
$string (
      value: any = $required
      format: key
          (elements, element, e)
          (data_structure, ds)
          (source, s)
        keyend = elements
      max_string: integer 1..$max_string = $max_string
      )
```

VALUE: This parameter specifies the value whose string representation is to be returned.

FORMAT: This parameter determines the form and content of the resulting string(s).

ELEMENTS, ELEMENT, E: Each element of the value is returned in a separate string. Status values are formatted as messages and may therefore occupy more than one string. The built-in record types command_reference, date_time, entry_point_reference, scu_line_identifier, time_increment, and time_zone are considerred to be elements in this format.

DATA_STRUCTURE, DS: Each element of the value is returned as a separate string. The generic data type of the value (and components of the value, if applicable) are include in the result as "comments" prefixing the corresponding value or value component. Items that are components of data structures are prefixed with identifying "labels," e.g. field names for record elements, subscripts for array elements, etc.

SOURCE, S: The returned string(s) contain an SCL "expression" that if evaluated with an appropriate type specification would yield the value. All result strings except the last will end with an ellipsis "..".

Omission causes ELEMENTS is used.

MAX_STRING: This parameter is used to constrain the length of the string(s) returned.

Omission causes the maximum string size is used (65535).

9.22.12 $substring

The $SUBSTRING function returns a portion of a string.

```
$substring (
      string: string = $required
      index: integer 1..$max_string+1 = $required
      size: integer 0..$max_string = 1
      fill: char = ' '
      )
```

STRING: This parameter specifies the string of which a substring is to be returned.

INDEX: This parameter specifies the first character of STRING to be included in the result.

SIZE: This parameter specifies the number of characters to be included in the result.

Omission causes one to be used..

FILL: This parameter is only used when SIZE is greater than the number of characters remaining in STRING and specifies the character used to pad the result to the requested length.

Omission causes the space character to be used.

9.22.13 $translate

The $TRANSLATE function is used to change characters in a string according to a translation table. The translation table is a string of 256 characters which is utilized according to the following algorithm.

```
          result = ''
          for i = 1 to $size(string) do
            result = result // table($ord(string(i))+1)
          forend


$translate (
      translation_table: any of
          recend
            base: key
              (lower_to_upper, ltu)
              (upper_to_lower, utl)
              (control_codes_to_question_marks, cctqm)
              none
            keyend
            original: string = $optional
            translated: string = $optional
          recend
          string 256
        anyend = $required
      string: string = $required
      )
```

TRANSLATION_TABLE: This parameter specifies the translation table. This table can be given as a 256 character string, a predefined table designated by a keyword, or a predefined (base) table modified by a set of replacement characters.

The predefined (BASE) translation tables are selected by the following keywords:

LOWER_TO_UPPER, LTU: produces a string with all lower case letters translated to their upper case counterparts.

UPPER_TO_LOWER, UTL: produces a string with all upper case letters translated to their lower case counterparts.

CONTROL_CODES_TO_QUESTION_MARKS, CCTQM: produces a string with all ASCII control code characters (i.e. any character whose ASCII ordinal is between 0 and 31, or 127 and 255) translated to the question mark character.

NONE: reproduces the original STRING, i.e. no translation occurs. This option is only useful when the ORIGINAL and TRANSLATED fields are also specified.

The ORIGINAL and TRANSLATED fields can be used to modify any of these predefined tables. If either of these fields is omitted, the BASE table is used unmodified. The modification is made by replacing entries in the BASE table such that a character appearing in ORIGINAL is translated to the corresponding character in TRANSLATED. If one of the ORIGINAL or TRANSLATED strings is longer than the other, the excess characters are ignored. If a character appears more than once in ORIGINAL, the rightmost translation is used.

STRING: This parameter specifies the string to be translated.

```
Example: display_value $translate(lower_to_upper, 'Brian')
         BRIAN
         display_value $translate((upper_to_lower, '_.:', './/'), ..
                                  ':NVE.JBF.COMMAND_LIBRARY')
         /nve/jbf/command.library
```

9.22.14 $trim

The $TRIM function is used to remove a trailing and/or leading characters from a string.

```
$trim (
      string: string = $required
      char: string 1 = ' '
      leading_or_trailing: list of key
          (trailing, t)
          (leading, l)
          all
        keyend = trailing
```

```
)
```

STRING: This parameter specifies the string to be trimmed.

CHAR: This parameter specifies the character to be trimmed from STRING.

Omission causes the space character to be used.

LEADING_OR_TRAILING: This parameter specifies whether LEADING (L) characters, TRAILING (T) characters, or both, i.e. ALL, should be trimmed.

Omission causes TRAILING characters to be trimmed.

```
Example: var
           s: string
         varend

         s = '     some stuff     '
         display_value '<'//s//'>'
         <     some stuff     >
         display_value '<'//$trim(s)//'>'
         <     some stuff>
         display_value '<'//$trim(s,,all)//'>'
         <some stuff>
```

9.23 Functions Related to String Patterns

A STRING_PATTERN is a data structure that controls the string matching process. It represents a pattern of characters which can be as simple as “any 3 characters” or “the string ‘abc,’” or can be very complex. The string pattern concepts within SCL are derived from those in the programming language SNOBOL4. The “wild card” facilities provided by SCL are implemented as a subset of string patterns.

The string matching process consists of attempting to find a sequence of characters in a *subject* string that are *matched* by a *pattern*. The result of this process may be *failure*, i.e. the pattern could not be found in the subject, or it may be *success*. In this case the *index* and *size* of the substring of the subject that matched the pattern can be made available.

A string pattern is composed of any number of *pattern elements*. These elements can be combined in essentially two ways: *concatenation* and *alternation*. When two pattern elements are concatenated, a match for the first must be found in the subject string immediately followed by a match for the second. The second of the two elements is said to be the *successor* to the first. The “//” operator is used to perform concatenation in string pattern expressions. See the section on *String Pattern Expressions* for more information.

When two pattern elements are combined via alternation, a match for either the first or the second must be found in the subject string. The second such element is said to be the *alternative* to the first. The $SP_ALTERNATION ($SP_OR) function, described below, produces alternation patterns.

The goal of the pattern matching process is to find a sequence of elements within a pattern that match a substring within the subject string.

There are two major steps involved in the use of string patterns. The first is to build the pattern, either from a description of what kinds of strings should be matched, or by combining existing patterns. The second step is to use the pattern to scan a subject string for a match.

The scanning step iterates through pattern elements following successors, advancing the subject index past the characters that matched the element, until an element fails to match or there are no more successors. In the latter case *success* is declared. If an element fails to match, the process *backs up* until an alternative is found, the subject index is reset to the point where the current element originally matched, then scanning continues by iterating through the alternative and its successors. When no more alternatives exists, *failure* is declared.

Patterns which are recursively defined can be represented using the *deferred* pattern element. A relatively common example of a recursive pattern is one that could match the arithmetic expression of a programming language, since such expressions can have sub-expressions enclosed in parentheses.

The pattern matching process is not, however, restricted to just looking at a subject string. With appropriate pattern

elements, substrings of the subject that match parts of the pattern can be captured for later use. The capture can be done either once the entire pattern has been successfully matched, or immediately upon matching the appropriate part of the pattern. The ability to immediately capture part of the subject just matched makes possible some very sophisticated matching. For example the captured substring could be referenced within an deferred pattern. This provides the capability to match patterns such as: “pattern A” followed by “pattern B” followed by whatever “pattern A” matched to the left of “pattern B.”

Attempted matches with some kinds of patterns won't have the intended results if the size information is used during the matching process. Therefore an option is provided that “shuts off” those checks. This “full scan” option should only be used when necessary since it can dramatically slow down pattern matching. The “quick scan” option should normally be used.

All of the functions that construct string pattern values have names of the form $SP_XXX, where XXX identifies the nature of the pattern. In the examples that appear in the following descriptions of these functions, other pattern building functions are often used. Therefore, to get the most out the descriptions of the functions, it may be useful to quickly read through all of the following subsections, then go back and re-read them more carfully.

9.23.1 $match

The $MATCH function returns the result of trying to match a pattern against a string. The result is in the form of a record whose type is:

```
record
  matched: boolean
  index: integer 1..$max_string+1
  size: integer 0..$max_string
recend
```

MATCHED: This field indicates whether the pattern could be matched within the subject (TRUE) or not (FALSE).

INDEX: If MATCHED is TRUE, this field indicates the starting position of the substring within the subject string that was matched by the pattern. If MATCHED is FALSE, this field is set to one.

SIZE: If MATCHED is TRUE, this field indicates the size of the substring within the subject string that was matched by the pattern. If MATCHED is FALSE, this field is set to zero.

```
$match (
      subject: string = $required
      pattern: string_pattern = $required
      anchor_option: key
          (anchored, a)
          (unanchored, u)
        keyend = unanchored
      scan_option: (advanced) key
          (quick_scan, qs)
          (full_scan, fs)
        keyend = quick_scan
      )
```

SUBJECT: This parameter specifies the string which is the subject of the match.

PATTERN: This parameter specifies the pattern to be matched.

ANCHOR_OPTION: This parameter specifies whether the matching process should be ANCHORED at the beginning of the subject string or UNANCHORED allowing the match to occur anywhere within the subject.

Omission causes UNANCHORED to be used.

SCAN_OPTION: This parameter specifies whether the heuristics used to optimize matching should be used (QUICK_SCAN) or not (FULL_SCAN). In general these two options will produce the same results but the QUICK_SCAN option allows the success or failure of the match to be determined more quickly for complex patterns.

The FULL_SCAN option can have different results for patterns involving “unconditional capture” or “deferred” pattern elements (see the $SP_CAPTURE and $SP_DEFER functions for more information).

Omission causes QUICK_SCAN to be used.

9.23.2 $sp_any

The $SP_ANY function returns a string pattern that matches any one of a specified set of characters.

```
$sp_any (
      characters: list rest of any of
          string



   range of string 1
        anyend = $required
      )
```

CHARACTERS: This parameter specifies the set of characters to be matched.

The set is described by a list of strings or ranges of characters. If a string is given, all of the characters in the string are included in the set. If a range of characters is given, all of the characters between and including those in the range, according to the ASCII collating sequence, are included in the set. (The order of the range is doesn't matter, i.e. '0'..'9' or '9'..'0' both specify all of the digit characters.)

```
Example: identifier = $sp_any('A'..'Z', 'a'..'z') // ..
                      $sp_repeat((any, 'A'..'Z', 'a'..'z', '_', '0'..'9'))
         for each x in ('abc', 'X_15', '_42', 'sp$hum', 'This_is_OK') do
           if $match(x, identifier//$sp_right, anchored).matched then
             display_value x//' is an identifier'
           else
             display_value x//' is NOT an identifier'
           ifend
         forend

         abc is an identifier
         X_15 is an identifier
         _42 is NOT an identifier
         sp$hum is NOT an identifier
         This_is_OK is an identifier
```

9.23.3 $sp_balance

The $SP_BALANCE function, returns a string pattern that matches any non-null string that is balanced with respect to a pair of characters, usually parentheses. $SP_BALANCE matches

X
XYZ
(A+B)
A(B*C)(E/F))G+H

and does not match

)A+B(
((A+B)

When first encountered $SP_BALANCE matches the shortest balanced string. If it backed into because one of its successors failed to match, it tries to increase the number of characters it matches.

```
$sp_balance (
      left_character: string 1 = '('
      right_character: string 1 = ')'
      )
```

LEFT_CHARACTER: This parameter specifies the left character of the pair to be balanced.

Omission causes a left parenthesis to be used.

RIGHT_CHARACTER: This parameter specifies the right character of the pair to be balanced.

Omission causes a right parenthesis to be used.

```
Example: "This somewhat contrived example shows how $SP_BALANCE operates."

         show_balancing = $sp_capture($sp_balance, unconditional, ..
                          'display_value', command) // $sp_fail
         subject = '((A+(B*C))+D)'
         ignore_match_result = $match(subject, show_balancing)

         ((A+(B*C))+D)
         (A+(B*C))
         (A+(B*C))+
         (A+(B*C))+D
         A
         A+
         A+(B*C)
         +
         +(B*C)
         (B*C)
         B
         B*
         B*C
         *
         *C
         C
         +
         +D
         D
```

9.23.4 $sp_capture

The $SP_CAPTURE function returns a string pattern that matches the pattern passed to it and “captures” the substring of the subject that it matches.

```
$sp_capture (
      pattern: string_pattern = $required
      when: key
          (conditional, c)
          (unconditional, immediate, i, u)
        keyend = $required
      where: string = $required
      how: key
          (variable, v)
          (command, c)
        keyend = variable
      )
```

PATTERN: This parameter specifies the pattern that, once matched, is to be “captured” as specified by the remaining parameters.

WHEN: This parameter specifies when the capture should occur. CONDITIONAL specifies that capturing should occur only if the entire pattern of which this element is a part is successfully matched. UNCONDITIONAL specifies that capturing should occur immediately upon successful matching of this part of the PATTERN.

UNCONDITIONAL capture is generally less efficient and should be avoided if it isn't needed. It can be useful in combination with the $SP_DEFER function and/or the FULL_SCAN matching option, in the latter case usually with

COMMAND specified for the HOW parameter.

WHERE: This parameter specifies the VARIABLE or COMMAND used to actually do the capturing.

HOW: This parameter specifies whether the capturing is to be done using a VARIABLE or a COMMAND.

If VARIABLE is specified, the matched substring is written to the variable specified by the WHERE parameter. If this variable does not already exist, it is created as a local string variable.

If COMMAND is specified, the matched substring is passed positionally as the first (only) parameter to the command specified by the WHERE parameter.

Omission causes VARIABLE to be used.

For an example of UNCONDITIONAL capture, see the example for the $SP_BALANCE function. For an example of CONDITIONAL capture, see the example for the $SP_COUNT function.

9.23.5 $sp_count

The $SP_COUNT function returns a string pattern that matches a specified number of characters.

```
$sp_count (
      count: integer 0..$max_string = $required
      not_any: list rest of any of
          string
          range of string 1
        anyend = $optional
      )
```

COUNT: This parameter specifies the number of characters to be matched

NOT_ANY: This parameter specifies the set of characters not to be matched.

The set is described by a list of strings or ranges of characters. If a string is given, all of the characters in the string are included in the set. If a range of characters is given, all of the characters between and including those in the range, according to the ASCII collating sequence, are included in the set. (The order of the range is doesn't matter, i.e. '0'..'9' or '9'..'0' both specify all of the digit characters.)

Omission allows all characters to be matched.

```
Example: split = $sp_capture($sp_count(6, ' '), conditional, 'first') // ..
                 $sp_count(3) // ..
                 $sp_capture($sp_upto(right), conditional, 'last')
         lines = ('Calvin : transmogrifier inventor', ..
                  'Hobbes : tiger', ..
                  'Fred doesn''t belong here.', ..
                  'Snoopy : dog')
         for each line in lines do
           if $match(line, split, anchored).matched then
             display_value first//' is a '//last//'.'
           else
             display_value line
           ifend
         forend

         Calvin is a transmogrifier inventor.
         Hobbes is a tiger.
         Fred doesn't belong here.
         Snoopy is a dog.
```

9.23.6 $sp_defer

The $SP_DEFER function returns a string pattern that matches a pattern supplied in the form of a string pattern expression. When this pattern is encountered during the matching process, the expression is evaluated and the resulting pattern is matched.

```
$sp_defer (
      pattern: (DEFER) string_pattern = $required
      minimum_match_size: integer 0..$max_string = 1
      )
```

PATTERN: This parameter specifies the deferred pattern.

MINIMUM_MATCH_SIZE: This parameter specifies the minimum number of characters that the PATTERN will match once it is evaluated. This number is used when the QUICK_SCAN option is selected for a matching operation. Providing a "good guess" for this number can help speed up the matching process.

```
Example: " This example shows a way of detecting a word that appears twice
         " in a row on line.

         identifier = $sp_any('A'..'Z', 'a'..'z') // ..
                      $sp_repeat((any, 'A'..'Z', 'a'..'z', '_', '0'..'9'))
         delimiter = $sp_repeat((not_any,'A'..'Z','a'..'z','_','0'..'9'),1)
         double = $sp_or($sp_left, delimiter) // ..
                  $sp_capture(identifier, unconditional, 'dv') // ..
                  delimiter // ..
                  $sp_defer(dv) // ..
                  $sp_or($sp_right, delimiter)

         lines = ('in the morning', ..
                  'in the, the morning', ..
                  'in in the morning', ..
                  'in the morning morning', ..
                  'in the early the morning')

         for each line in lines do
           result = $match(line, double)
           display_value '<'//line(result.index, result.size)//'>'
         forend

         <>
         < the, the >
         <in in >
         < morning morning>
         <>

Example: " This example shows a recognizer for simple arithmetic expressions.

         variable   = $sp_any('A'..'Z', 'a'..'z') // ..
                      $sp_repeat((any, 'A'..'Z', 'a'..'z', '_', '0'..'9'))
         constant   = $sp_repeat((any, '0'..'9'), 1)
         add_op     = $sp_any('+-')
         mul_op     = $sp_any('*/')
         factor     = $sp_or(constant, variable, ..
                             '('//$sp_defer(expression)//')')
         term       = factor // $sp_repeat(mul_op//factor)
         expression = $sp_or(add_op, $sp_null) // term // ..
                      $sp_repeat(add_op//term)

         candidates = ('x+y*(z+x)', ..
```

```
                '-count+23', ..
                'what is this')

for each c in candidates do
  if $match(c, expression//$sp_right, anchored).matched then
    display_value c//' is an expression'
  else
    display_value c//' is not an expression'
  ifend
forend

x+y*(z+x) is an expression
-count+23 is an expression
what is this is not an expression
```

9.23.7 $sp_fail

The $SP_FAIL function returns a string pattern that always fails to match, causing the matching process to back up and seek alternatives. I.e. it provides a way to build a structure that does *not* match the pattern to which this element is a successor.

```
$sp_fail (
      )
```

A common use for $SP_FAIL is shown in the example for the $SP_BALANCE function. There it causes an otherwise successful match to fail, in turn causing alternatives to be tried. This technique can be used to observe the behavior of the pattern matching process. Also, the example for the $SP_SUCCEED function.

9.23.8 $sp_fence

The $SP_FENCE function returns a string pattern that matches the null string. If it is backed into because one of its successors fails, it causes immediate failure termination of the entire matching process. It can be used to avoid trying alternatives that can't possibly match.

```
$sp_fence (
      )
```

For example, given the following

```
pat = $sp_or('a', 'the') // ' mousetrap'
str = 'a fleatrap'
```

$match(str, pat, anchored) will fail, but only after attempting to match `'the'` at the beginning of `str`. There is no point in trying this second alternative, since `'the'` can't possibly match what `'a'` matched. For this kind of pattern, avoiding fruitless alternatives can make the matching process report failure more quickly. For example, if `pat` is changed to

```
pat = $sp_or('a', 'the') // $sp_fence // ' mousetrap'
```

the match will still fail, but more quickly. When the $SP_FENCE is encountered while backing up to try the alternative to `'a'`, the matching process immediately stops.

9.23.9 $sp_index

The $SP_INDEX function returns a string pattern that matches the null string and “captures” the current value of the scanning (matching) index into the subject string.

```
$sp_index (
      where: string = $required
      how: key
          (variable, v)
          (command, c)
        keyend = variable
      )
```

WHERE: This parameter specifies the VARIABLE or COMMAND used to actually do the capturing.

HOW: This parameter specifies whether the capturing is to be done using a VARIABLE or a COMMAND.

If VARIABLE is specified, the index is written to the variable specified by the WHERE parameter. If this variable does not already exist, it is created as a local integer variable.

If COMMAND is specified, the matched substring is passed positionally as the first (only) parameter to the command specified by the WHERE parameter. The index is passed as a string.

Omission causes VARIABLE to be used.

```
Example: " In this example $SP_INDEX and $SP_TEST are used to restrict a
         " pattern to match prior to column 10 of a subject.

         position_check = $sp_index('pos') // $sp_test(pos<=10)
         pat = $sp_any('0'..'9')
         restricted_pat = pat // position_check

         lines = ('12 days to go', ..
                  'you have at least 6 weeks', ..
                  'in 3 years time')

         for each line in lines do
           if $match(line, restricted_pat, unanchored).matched then
             display_value line
           ifend
         forend

         12 days to go
         in 3 years time
```

9.23.10 $sp_left

The $SP_LEFT function returns a string pattern that matches the null string if there are exactly COUNT characters to the left of the subject index. This is most frequently used with a COUNT of zero in order to force its successor to be matched at the left end of the subject.

```
$sp_left (
      count: integer 0..$max_string = 0
      )
```

COUNT: This parameter specifies the required number of characters to the left of the subject index. Giving zero for this parameter has the same effect as selecting the ANCHORED option when performing a match operation.

Omission causes zero to be used.

```
Example: first_six_non_blank = $sp_upto(' ') // $sp_left(6)

         lines = ('123456 good', ..
                  'abcdef  not bad', ..
                  '987654321 to many', ..
                  'XY not enough')
```

```
          for each line in lines do
            if $match(line, first_six_non_blank, anchored).matched then
              display_value line//'...is OK'
            else
              display_value line//'...has an "early" or "late" blank'
            ifend
          forend

          123456 good...is OK
          abcdef  not bad...is OK
          987654321 to many...has an "early" or "late" blank
          XY not enough...has an "early" or "late" blank
```

9.23.11 $sp_not_any

The $SP_NOT_ANY function returns a string pattern that matches any one character that is not in a specified set of characters.

```
$sp_not_any (
      characters: list rest of any of
          string
          range of string 1
        anyend = $required
      )
```

CHARACTERS: This parameter specifies the set of characters not to be matched.

The set is described by a list of strings or ranges of characters. If a string is given, all of the characters in the string are included in the set. If a range of characters is given, all of the characters between and including those in the range, according to the ASCII collating sequence, are included in the set. (The order of the range is doesn't matter, i.e. '0'..'9' or '9'..'0' both specify all of the digit characters.)

```
Example: " The following pattern matches any string that does not contain
          " exactly one digit.

          not_one_digit = $sp_left // $sp_not_any('0'..'9') // $sp_right
```

9.23.12 $sp_null

The $SP_NULL function returns a string pattern that matches the null string.

```
$sp_null (
      )
```

```
Example: name = $sp_repeat((any, 'A'..'Z', 'a'..'z'), 1)
          number = $sp_repeat((any, '0'..'9'), 1)
          delimiter = $sp_any(' ,:-')
          optional_delimiter = $sp_or($sp_null, delimiter)
          check_name_number = $sp_left // ..
                              $sp_capture(name, conditional, 'who') // ..
                              optional_delimiter // ..
                              $sp_capture(number, conditional, 'n') // ..
                              $sp_right

          lines = ('Robyn:13', ..
                   'Donna 5', ..
```

```
                'Tom,1200', ..
                'Carol-999', ..
                'John42', ..
                'oh oh')

        for each line in lines do
          if $match(line, check_name_number).matched then
            display_value who//': '//n
          else
            display_value 'Dont''t understand: '//line
          ifend
        forend

        Robyn: 13
        Donna: 5
        Tom: 1200
        Carol: 999
        John: 42
        Dont't understand: oh oh
```

9.23.13 $sp_or

The $SP_OR returns a string pattern that matches any of patterns passed to it.

```
$sp_or (
      patterns: list rest of string_pattern = $required
      )
```

PATTERNS: This parameter specifies the patterns to be joined as alternatives. The order in which these patterns are specified determines the order in which they are tried during a match operation.

```
Example: it = $sp_or('file', 'catalog')
         it_count = 0
         lines = ('NOS/VE provides files' ..
                  'and catalogs which can' ..
                  'contain them.')
         for each line in lines do
           it_count = it_count + $integer($match(line, it).matched)
         forend
         display_value it_count

         2
```

9.23.14 $sp_repeat

The $SP_REPEAT function returns a string pattern that matches a pattern repeatedly. When first encountered, it tries to match the pattern a specified minimum number of times. When “backed into” because one of its successors failed to match, it tries to extend what it matched

```
$sp_repeat (
      what: any of
          record
            any_or_not_any: key
              any
              not_any
            keyend
            characters: list rest of any of
```

```
            string
            range of string 1
          anyend = $optional
        recend
        string_pattern
      anyend = $required
    minimum_count: integer 0..$max_string = 0
    )
```

WHAT: This parameter specifies the pattern to be repeatedly matched.

If the ANY keyword is specified and the CHARACTERS set is omitted, the pattern matches MINIMUM_COUNT or more characters.

If the ANY keyword is specified and the CHARACTERS set is also specified, the pattern matches MINIMUM_COUNT or more of the specified characters.

If the NOT_ANY keyword is specified, this element matches MINIMUM_COUNT or more characters. Any character specified by the CHARACTERS set is not matched.

For the previous two cases, the set of characters is described by a list of strings or ranges of characters. If a string is given, all of the characters in the string are included in the set. If a range of characters is given, all of the characters between and including those in the range, according to the ASCII collating sequence, are included in the set. (The order of the range is doesn't matter, i.e. '0'..'9' or '9'..'0' both specify all of the digit characters.)

If a pattern is specified, it is matched at least MINIMUM_COUNT times.

MINIMUM_COUNT: This parameter specified the minimum number of times the repeated pattern must be matched.

Omission causes zero to be used.

The pattern `$sp_repeat((any, '0'..'9'), 1)` matches an unsigned decimal integer:

The pattern `$sp_repeat((not_any, '.;'))` matches a sequence of chaaracters that do not contain a period or semicolon.

If `number` has been assigned the pattern `$sp_repeat((any, '0'..'9'), 1)`, then the pattern

```
number // $sp_repeat(','//number)
```

matches a comma-separated list of numbers.

$SP_REPEAT is used extensively in examples for other string pattern functions.

9.23.15 $sp_right

The $SP_RIGHT function returns a string pattern that matches the null string if there are exactly COUNT characters to the right of the subject index. This is most frequently used with a COUNT of zero in order to force its predecessor to be matched at the right end of the subject.

```
$sp_right (
      count: integer 0..$max_string = 0
      )
```

COUNT: This parameter specifies the required number of characters to the right of the subject index.

Omission causes zero to be used.

As mentioned above, $SP_RIGHT is most often used to ensure that its predecessor is the last, or only, thing in the subject. For example, if `number` has been assigned the pattern `$sp_repeat((any, '0'..'9'), 1)`, then the pattern `number//$sp_right` matches any string that ends with a sequence of digits. And the pattern `$sp_left//number//$sp_right` matches any string that contains only digits.

9.23.16 $sp_stop

The $SP_STOP function returns a string pattern that causes immediate failure termination of the entire matching process.

```
$sp_stop (
      )
```

$SP_STOP can be used to build “conditional” patterns. For example, given two patterns, P and Q, the following pattern matches a string that Q doesn't match, but P does:

```
P_NOT_Q = $sp_or(Q//$sp_stop, P)
```

If Q matches the subject, $SP_STOP is encountered and matching immediately fails. If Q fails to match, the alternative P is tried.

9.23.17 $sp_string

The $SP_STRING function returns a string pattern that matches the specified string of characters. This function can be used to explicitly convert a string to the pattern that matches it.

```
$sp_string (
      string: string = $required



case_option: key
          (case_sensitive, cs)
          (ignore_case, ic)
        keyend = case_sensitive
      )
```

STRING: This parameter specifies the string to be matched.

CASE_OPTION: This parameter specifies whether the case of characters matters when STRING is matched. If CASE_SENSITIVE is used, the STRING must match exactly as given. If IGNORE_CASE is zpecified, any lower case characters in STRING and the subject string being matched are, in effect, “folded” to their upper case counterparts prior to attempting the match. The “folding” is done according to the current value of the SCL option FOLDING_LEVEL.

Omission causes CASE_SENSITIVE to be used.

For example, `$sp_string('copy',ignore_case)` matches `copy`, `COPY`, `Copy`, `CoPy`, etc., whereas `$sp_string('copy')` (or simply `'copy'`) just matches `copy`.

9.23.18 $sp_succeed

The $SP_SUCCEED function returns a string pattern that matches the null string. If it is backed into because one of its successors fails, it succeeds again, i.e. it can be thought of as being its own alternative.

```
$sp_succeed (
      )
```

Its usefulness is limited but, in combination with the UNCONDITIONAL option of the $SP_CAPTURE function and the $SP_FAIL function, it can produce “interesting” results.

```
Example: sawtooth_pattern = $sp_succeed // ..
                            $sp_capture($sp_repeat(any, 1), unconditional, ..
                                        'display_value', command) // ..
                            $sp_fail
```

```
sawtooth_subject = 'XXXXXX'

ignore = $match(sawtooth_subject, sawtooth_pattern)

X
XX
XXX
XXXX
XXXXX
XXXXXX
X
XX
XXX
XXXX
XXXXX
XXXXXX
X
XX

... and so on forever
```

In the example above, $SP_FAIL prevents the pattern from ever matching successfully and $SP_SUCCEED prevents it from failing. Therefore the matching process continually bounces back and forth between these two functions.

9.23.19 $sp_test

The $SP_TEST function returns a string pattern that tests the value of a boolean expression when it is encountered during pattern matching. If the expression evaluates to TRUE, this pattern matches the null string. If the expression evaluates to FALSE or can't be evaluated, this pattern fails.

```
$sp_test (
      test: (DEFER) boolean = $required
      )
```

TEST: This parameter specifies the boolean expression.

An example of $SP_TEST can be found in the section on the $SP_INDEX function.

9.23.20 $sp_upto

The $SP_UPTO function returns a string pattern that matches everything up to a specified point in the subject string.

```
$sp_upto (
      where: any of
          key
            (left, l)
            (right, r)
          keyend
          list of any of
            string
            range of string 1
          anyend
        anyend = $required
      count: integer 0..$max_string = 0
      )
```

WHERE: This parameter specifies the point in the subject to match up to.

If LEFT is specified, this pattern matches from the current position in the subject up to and including the COUNT'th

character from the left end of the subject.

If RIGHT is specified, this pattern matches from the current position in the subject up to and including the COUNT'th character from the right end of the subject. This is most frequently used with a COUNT of zero in order to match the rest of the subject.

If a set of characters is specified, this pattern matches from the current position in the subject up to but not including one of the characters contained in the set.

The set is described by a list of strings or ranges of characters. If a string is given, all of the characters in the string are included in the set. If a range of characters is given, all of the characters between and including those in the range, according to the ASCII collating sequence, are included in the set. (The order of the range is doesn't matter, i.e. '0'..'9' or '9'..'0' both specify all of the digit characters.)

COUNT: This parameter specifies the number of characters from the left or right end of the subject. It is ignored if a set of characters is specified for the WHERE parameter.

Omission causes zero to be used.

```
Example: " In this example a pattern is used to break input lines into three
         " fields.  The first field is everything upto the first blank.  The
         " second field is everything from the first blank upto column six.
         " The third field is everthing from column 7 to the end of the line.
         " Lines that are successfully matched are displayed with the fields
         " reordered so that the contents of field one are now right justified
         " rather than left justified.

         break_up = $sp_capture($sp_upto(' '), conditional, 'f1') // ..
                    $sp_capture($sp_upto(left, 6), conditional, 'f2') // ..
                    $sp_capture($sp_upto(right), conditional, 'f3')

         lines = ('234   miles', ..
                  '7000  degrees', ..
                  'an odd one', ..
                  'un-break_up-able line', ..
                  '42    is the answer')

         for each line in lines do
           if $match(line, break_up).matched then
             display_value f2//f1//' '//f3
           else
             display_value line
           ifend
         forend

            234 miles
           7000 degrees
          oddan  one
         un-break_up-able line
             42 is the answer
```

9.23.21 $sp_wild_card, $sp_wc

The $SP_WILD_CARD function returns a string pattern that matches a wild card pattern.

```
$sp_wild_card, $sp_wc (
      pattern: (wild_card_pattern) any of
          string
          application
        anyend = $required
      pattern_type: key
```

```
          (basic, b)
          (extended, e)
        keyend = $scl_options.wild_card_pattern_type
      )
```

PATTERN: This parameter specifies the wild card pattern.

PATTERN_TYPE: This parameter specifies the whether the PATTERN should be interpreted as a BASIC or EXTENDED wild card pattern. (See the section on *Wild Card Patterns* for more information.)

Omission causes the SCL option WILD_CARD_PATTERN_TYPE to be used.

```
Example: change_scl_option wild_card_pattern_type=extended "the default"

         display_value $sp_wild_card(??[cmptv]$?*)

         $sp_count(2)//$sp_any('c','m','p','t','v')//'$'//$sp_repeat(any,1)

         display_value $sp_wild_card({source*|*library})

         $sp_or('source'//$sp_repeat(any),$sp_repeat(any)//'library')
```

9.24 Functions Related to Types

9.24.1 $element_type

The $ELEMENT_TYPE function is passed a variable of type array, list or range and returns the type of the element of the structure.

```
$element_type (
      structured_variable: data_name = $required
      )
```

STRUCTURED_VARIABLE: This parameter specifies the variable whose element type is to be returned.

9.24.2 $generic_type

The $GENERIC_TYPE function returns the name of the generic type of an expression.

```
$generic_type (
      data: any = $required
      )
```

DATA: This parameter specifies the value whose generic type is to be returned. The name returned is the first word that appears in the declaration of the type, i.e. the function result type is described by:

```
         key
           application
           array
           boolean
           COBOL_name
           command_reference
           data_name
           date
           date_time
```

```
      entry_point_reference
      file
      integer
      key
      line_identifier
      list
      lock
      name
      network_title
      program_name
      range
      real
      record
      statistic_code
      status
      status_code
      string
      string_pattern
      time
      time_increment
      time_zone
      type
    keyend
```

9.24.3 $lower_value

The $LOWER_VALUE function is passed a type specification for an integer type and returns the lowest allowed value for that type.

```
$lower_value (
      integer_type: type = $required
      )
```

INTEGER_TYPE: This parameter specifies the integer type specification.

9.24.4 $max_string_size

The $MAX_STRING_SIZE function is passed a type specification for a string type and returns the maximum size attribute for that type.

```
$max_string_size (
      string_type: type = $required
      )
```

STRING_TYPE: This parameter specifies the string type specification.

9.24.5 $min_string_size

The $MIN_STRING_SIZE function is passed a type specification for a string type and returns the minimum size attribute for that type.

```
$min_string_size (
      string_type: type = $required
      )
```

STRING_TYPE: This parameter specifies the string type specification.

9.24.6 $type

The $TYPE function returns the type of the variable that is its parameter.

```
$type (
      variable: data_name = $required
      )
```

VARIABLE: This parameter specifies the variable whose type is to be returned.

9.24.7 $upper_value

The $UPPER_VALUE function is passed a type specification for an integer type and returns the highest allowed value for that type.

```
$upper_value (
      integer_type: type = $required
      )
```

INTEGER_TYPE: This parameter specifies the integer type specification.

9.25 Functions Related to Variables

9.25.1 $variable

The $variable function is passed a variable name and an attribute keyword, and returns a boolean value indicating whether the variable has the specified attribute.

```
$variable (
      variable: data_name = $required
      attribute: key
          defined
          environment
          local
          nonlocal
          read
          initialized
        keyend = $required
      )
```

VARIABLE: This parameter specifies the variable to be interrogated.

ATTRIBUTE: This parameter specifies the attribute to be interrogated. (See the section on “functions related to types” for a description of functions that can be used to interrogate a variable's type.)

DEFINED: This attribute is true if any of LOCAL, NONLOCAL or ENVIRONMENT attributes is true.

ENVIRONMENT: This attribute is true if the variable is an environment variable that was defined in or is accessible from the requesting block.

LOCAL: This attribute is true if the variable was defined in the current block.

NONLOCAL: This attribute is true if the variable was not defined in but is accessible from the requesting block. A NONLOCAL procedure variable, i.e. one declared with the XDCL attribute, requires an XREF declaration in the requesting block before it can be accessed.

READ: This attribute is true if the variable has the read-only attribute.

INITIALIZED: This attribute is true if the variable has been given a value.

9.25.2 $vname

The $vname function provides a means for indirectly referring to a variable. The string passed to this function is interpreted as a <variable>.

```
$vname (
      variable: string = $required
      )
```

VARIABLE: This parameter specifies the string to be treated as a variable reference.

9.26 Miscellaneous Functions

9.26.1 $cpu_time

The $cpu_time function returns the number of microseconds of central processing unit (CPU) time spent in a job or a task in job and/or monitor mode.

```
$cpu_time (
      accumulator: key
          (job, j)
          (job_job_mode, jjm)
          (job_monitor_mode, jmm)
          (task, t)
          (task_job_mode, tjm)
          (task_monitor_mode, tmm)
        keyend = job
      )
```

ACCUMULATOR: This parameter specifies which time accumulator is to be returned.

JOB, J: the sum of JOB_JOB_MODE and JOB_MONITOR_MODE

JOB_JOB_MODE, JJM: the CPU time spent in job mode in the requesting job

JOB_MONITOR_MODE, JMM: the CPU time spent in monitor mode in the requesting job

TASK, T: the sum of TASK_JOB_MODEand TASK_MONITOR_MODE

TASK_JOB_MODE, TJM: the CPU time spent in job mode in the requesting task

TASK_MONITOR_MODE, TMM: the CPU time spent in monitor mode in the requesting task

9.26.2 $job_termination_status

The $JOB_TERMINATION_STATUS can be called during epilog processing to determine the job's termination status. It has no parameters.

```
$job_termination_status (
      )
```

9.26.3 $mainframe

The $MAINFRAME function interrogates the attributes of the hardware mainframe on which the request is made. The kind of result returned by the function depends on the particular attribute being tested.

```
$mainframe (
      attribute: key
          (active_processors, active_processor, ap)
          (clock, c)
          (identifier, id, i)
          (page_size, ps)
          (total_processors, total_processor, tp)
        keyend = $required
      )
```

ATTRIBUTE: This parameter specifies the attribute to be returned.

ACTIVE_PROCESSORS, ACTIVE_PROCESSOR, AP: the number of processors that are 'ON' in the mainframe

CLOCK, C: the current value of the mainframe's free running microsecond clock

IDENTIFIER, ID, I: a string is returned which designates the mainframe (e.g., '$SYSTEM_0860_0209')

PAGE_SIZE, PS: the number of bytes in a memory page

TOTAL_PROCESSORS, TOTAL_PROCESSOR, TP: the number of processors that are in the mainframe in any state

9.26.4 $processor

The $PROCESSOR function interrogates the attributes of a hardware processor in the mainframe on which the request is made. The kind of result returned by the function depends on the particular attribute being tested.

```
$processor (
      attribute: key
          (model, m)
          (model_number, mn)
          (model_type, mt)
          (serial_number, serial, sn)
          (state, s)
        keyend = $required
      processor_number: integer 0..1
      )
```

ATTRIBUTE: This parameter specifies the attribute to be returned.

MODEL, M: a string is returned which designates the processor model type and number (e.g., 'CYBER 180-860')

MODEL_NUMBER, MN: a string is returned which designates the processor model number (e.g., '860')

MODEL_TYPE, MT: a string is returned which designates the processor model type (e.g., 'CYBER 180')

SERIAL_NUMBER, SN: the processor's serial number is returned as a string (e.g., '109')

STATE, S: the processor's state is returned as a string; — 'ON' (the processor is active), 'OFF' and 'DOWN'

PROCESSOR_NUMBER: This parameter specifies which processor of a multiprocessor mainframe is being tested. If the requested processor does not exist, a null string is returned for any attribute other than CLOCK.

Omission causes the current processor to be used.

9.26.5 $read (*)

The $READ function is used to read a file and a return a value corresponding to the data read. The interpretation of the data is determined by the context in which the function is called. It determines the data type it is expected to return and uses

that type specification to evaluate the data from the file.

One item is read from the file if the expected return type is an “elemental” type. If the expected return type is a list type, each item in the file is interpreted as an element of the list. If the expected return type is an array, an item is read for each element of the array. If the expected return type is a record, an item is read for each field of the record.

The minimum amount of data read from the file is a line, even though less than a line may actually be used.

```
$read (
      file: file = $required
      item_delimiter: any of
          key
            (end_of_line, eol)
          keyend
          string 1
        anyend = end_of_line
      )
```

FILE: This parameter specifies the file to be read.

ITEM_DELIMITER: This parameter determines what constitutes an item of data from the file. A character can be specified that separates items on the same line from one another or the keyword END_OF_LINE (EOL) specifies that each line is an item.

Omission causes END_OF_LINE to be used.

9.26.6 $system

The $SYSTEM function is used to interrogate various attributes of the system on which the request is made.

```
$system (
      attribute: key
          (catalog, c)
          (dual_state_partner, dsp) (*)
          (version, v) (*)
        keyend = catalog
      )
```

ATTRIBUTE: This parameter specifies the attribute to be returned.

CATALOG, C: a value of type file is returned which designates the catalog containing system supplied files and subcatalogs (i.e. :$SYSTEM.$SYSTEM)

DUAL_STATE_PARTNER, DSP (*): a string is returned which represents the name of the NOS/VE system's dual state partner system ('NOS' or 'NOS/BE'). If NOS/VE is running as a stand-alone system, i.e. without a dual state partner system, a null string is returned

VERSION, V (*): a string is returned which represents the name and version of the system

Omission causes CATALOG to be used.

9.26.7 $unique

The $UNIQUE function returns a name which is unique within the context of all mainframes running NOS/VE.

```
$unique (
      result: any of
          key
            string
            name
          keyend
          file
```

```
        anyend = string
      )
```

RESULT: This parameter selects the form in which the result is to be returned: as a STRING, a NAME or a file. For the file option, a catalog path is specified and a unique name is appended to that path.

Omission causes STRING to be assumed.

```
Example: display_value $unique do=data_structure
         "STRING"  '$708608230S0109D19880711T230606'
         display_value $unique(name) do=data_structure
         "NAME"  $708608230S0109D19880711T230606
         display_value $unique($local) do=data_structure
         "FILE"  :$LOCAL.$708608230S0109D19880711T230606
```

9.26.8 $use_value

The $USE_VALUE function returns a value of a specified TYPE. The value is eitherr that of a VARIABLE, if the variable exists and has a value of the appropriate type, or the value of an EXPRESSION.

```
$use_value (
      type: type = $required
      variable: data_name = $required
      expression: (defer) any = $required
      )
```

TYPE: This parameter specifies the type of value to be returned.

VARIABLE: This parameter specifies the name of the variable. If this variable exists and its value conforms to TYPE, its value becomes the result of the function.

EXPRESSION: This parameter specifies an expression to be evaluated if the VARIABLE does not exist or is not of the desired type. The result value of the expression becomes the result of the function.

An example of the use of this function is in a program description such as the following:

```
      create_program_description n=show_stuff sp=start ..
            l=:$defer.$use_value(file, plot_library, .libraries.plotting)
```

PLOT_LIBRARY specifies the name of a FILE variable whose value is used if the variable exists. Otherwise the FILE specified by .LIBRARIES.PLOTTING is used.

10 Structured Statements

Structured statements are optionally labelled constructs providing control over their constituent statement lists.

10.1 Statement Labels

```
<label> ::= <name>

<start label> ::= <label> <:> [<sp>]
                | <label> <sp> <:> <sp>
```

Labels provide a means of referring to a structured statement. A label precedes the statement it labels and is separated from it by a colon.

The scope of a label is the statement it names. It is impossible to refer to a label on a structured statement from outside that statement. A label may optionally follow a structured statement (except for the repeat statement), in which case it must be identical to the label at the start of that statement. This is for checking purposes only and does not affect the meaning of the statement. The scope of the label does not extend to procedures called from within its scope or to statement lists introduced by INCLUDE_FILE or INCLUDE_LINE commands.

10.2 BLOCK / BLOCKEND

```
<block statement> ::= [<start label>] BLOCK <;|nl>
                        [<statement list>]
                      BLOCKEND [<sp> <label>]
```

Block statements provide for grouping together a sequence of statements. Exit from the block is either through completing execution of the last statement of the statement list or through explicit transfer of control (e.g. an EXIT statement).

10.3 LOOP / LOOPEND

```
<loop statement> ::= [<start label>] LOOP <;|nl>
                       [<statement list>]
                     LOOPEND [<sp> <label>]
```

The LOOP statement provides for unlimited repetition of its statement list. Exit from a LOOP statement is only possible via an explicit transfer of control (e.g. an EXIT statement).

10.4 WHILE / WHILEND

```
<while statement> ::= [<start label>] WHILE <sp> <boolean expression> <do>
                        <statement list>
                      WHILEND [<sp> <label>]

<do> ::= [<sp> DO] <;|nl>
```

The WHILE statement provides for conditional repetition of its statement list. Prior to each iteration of the <statement list> the <boolean expression> is evaluated. If it is true the <statement list> is executed, otherwise control passes to the statement following the WHILEND.

10.5 REPEAT / UNTIL

```
<repeat statement> ::= [<start label>] REPEAT <;|nl>
                         <statement list>
                       UNTIL <sp> <boolean expression>
```

The REPEAT statement provides for conditional repetition of its statement list. After each iteration of the <statement list> the <boolean expression> is evaluated. If it is false the <statement list> is executed again, otherwise control passes to the statement following the UNTIL.

10.6 FOR / FOREND

```
<for statement> ::= [<start label>] FOR <sp> <for control> <do>
                      [<statement list>]
                    FOREND [<sp> <label>]

<for control> ::= <for increment control>
                | <for list control>

<for increment control> ::= <variable> <=> <initial> <sp> TO <sp> <final>
                            [<sp> BY <sp> <step>]

<initial> ::= <integer expression>
<final> ::= <integer expression>
<step> ::= <integer expression>

<for list control> ::= EACH <sp> <variable> <sp> IN <sp> <list expression>
```

The FOR statement provides controlled repetition of its statement list. Two methods of repetition are provided: incremental control via an integer variable, and sequential accessing of the elements of a list.

The variable following the word FOR or EACH is called the control variable of the FOR statement. Any alteration of the control variable within the statement list has no effect on the operation of the FOR statement.

10.6.1 Incremental Control

If the control variable isn't defined it is created as a local variable of type INTEGER.

Any alteration of <initial>, <final> or <step> within the statement list has no effect on the operation of the FOR statement.

The incremental control form of the FOR statement is illustrated by the SCL statements in the left column of the following table. In this illustration $1, $2 and $3 designate internal variables used by SCL to process the FOR statement. The right column shows the corresponding components of the FOR statement.

$1 = <initial> $2 = <final> $3 = <step> "1 if BY <step> omitted" WHILE (($3 >= 0) AND ($1 <= $2)) OR ((($3 < 0) AND ($1 >= $2)) DO <variable> = $1	FOR <variable> = <initial> .. TO <final> .. [BY <step>]
<statement list>	<statement list>
WHILEND	FOREND

10.6.2 List Control

If the control variable isn't defined it is created as a local variable whose type is determined by the element type of the <list expression>.

Any alteration of <list expression> within the statement list has no effect on the operation of the FOR statement.

The operation of the list control form of the FOR statement is illustrated by the SCL statements in the left column of the following table. In this illustration $1 designates an internal variable used by SCL to process the FOR statement. The right column shows the corresponding components of the FOR statement.

$1 = <list expression>
WHILE NOT $nil($1) DO
  <variable> = $first($1)
  $1 = $rest($1)
	FOR EACH <variable> ..
	  IN <list expression> DO

<statement list>	<statement list>
WHILEND	FOREND

10.7 IF / ELSEIF / ELSE / IFEND

```
<if statement> ::= IF <sp> <boolean expression> <then>
                     [<statement list>]
                   [ELSEIF <sp> <boolean expression> <then>
                     [<statement list>]]...
                   [ELSE <;|nl>
                     [<statement list>]]
                   IFEND

<then> ::= [<sp> THEN] <;|nl>
```

The IF statement provides for conditional execution of one of its constituent statement lists. The boolean expressions on the IF and ELSEIF clauses are evaluated in turn until one with a TRUE value is found or until an

ELSE clause is found. The statement list that follows is then executed after which control passes to the statement following the IFEND.

10.8 CASE / <Case Selection> / ELSE / CASEND (*)

```
<case statement> ::= CASE <sp> <expression> [<sp> OF] <;|nl>
                       <case selection> <;|nl>
                         [<statement list>]
                       [<case selection> <;|nl>
                         [<statement list>]]...
                       [ELSE <;|nl>
                         [<statement list>]]
                       CASEND

<case selection> ::= = [<sp>] <case selectors> [[<sp>] =]

<case selectors> ::= <case selector> [<,|sp> <case selector>]...

<case selector> ::= <case selector range>
                  | <case selector item>

<case selector range> ::= <expression> <..> <expression>

<case selector item> ::= <expression>
```

The CASE statement provides for conditional execution of one of its constituent statement lists. The expression in the CASE clause of the statement is evaluated and the result is used to search for a <case selection> that matches. A <case selection> matches the result if the result is equal to the value of one of its <case selector item>s, or if the result is within the range specified by one of its <case selector range>s. A value is considerred to be within a range if is greater than or equal to the first expression in the range and less than or equal to the second expression in the range.

If no matching <case selection> is found, the ELSE clause, if present, is considerred to match.

The statement list that follows the matching <case selection> or ELSE clause is executed after which control passes to the statement following the CASEND.

If no match is found and an ELSE clause is not present, control passes to the statement following the CASEND.

The order of searching the <case selector>s is not guaranteed. This implies that if more than one <case selection> contains a match for the CASE expression, the SCL interpreter arbitrarily chooses one of the candidates.

11 Error Handling

Error conditions can be dealt with in a number of ways within SCL as described in the following subsections.

11.1 Status Parameters

All commands have, by convention, an optional parameter called STATUS. This parameter specifies a variable of type STATUS that can be used to capture the completion status of the command on which it is specified. The parameter's presence or absence determines the action taken by the SCL interpreter if the command terminates with a status whose severity is ERROR or above.

If the status parameter is omitted from a command that terminates with a status whose severity is ERROR or above (and in the absence of an appropriate condition handler—see below), processing of the file, procedure or batch job containing the command is terminated with that same status. For commands entered interactively, prompts are issued for corrections to erroneous parameters.

For a command issued from the main command stream of a job, interactive or batch, that terminates abnormally, its status message is written to the $RESPONSE file if its status parameter was omitted.

If the status parameter is given, processing continues with the next statement or command even if the command terminated abnormally. The SCL interpreter presumes the command's completion status will be checked or ignored, as appropriate, by subsequent statements.

Certain kinds of errors in the call to the command can prevent this status parameter mechanism from producing the desired result. For example, if no processor for the command can be located, the parameters for the command are not examined since it is the command processor that informs the SCL interpreter of how to evaluate the command's parameters. Also, if for some reason the status parameter cannot be recognized, the processing described above cannot take place. Something that might prevent the status parameter from being recognized is leaving off the terminating apostrophe (quote mark) from a string constant that preceded the intended status parameter.

11.2 Conditions and Condition Handlers

To deal with those error situations that the status parameter mechanism cannot adequately address, SCL provides the condition handler mechanism. This mechanism is also used to handle “unexpected” conditions that are not necessarily errors.

A condition handler is established by means of the WHEN/WHENEND statement. The purpose of the condition handler is to deal with the conditions defined in the WHEN clause of the WHEN/WHENEND statement by means of the statement list of the WHEN/WHENEND statement.

What happens upon completion of a condition handler is determined by a CONTINUE or EXIT statement executed within the handler. If neither of these statements is used, CONTINUE NEXT is assumed.

11.2.1 Classes of Conditions

There are four “classes” of conditions: fault, informative, exit, and debug.

Fault conditions arise when an (unexpected) error occurs. They have an associated status describing the particulars of the fault.

Informative conditions represent asynchronous events for which special handling may be desired.

The special condition name ANY_FAULT is used to establish a condition handler to deal with any fault condition. The special condition name ANY_CONDITION is used to establish a condition handler to deal with any fault or informative condition.

Exit conditions are arguably the single most useful class of conditions. They provide the means to ensure that SCL procedures, utilities, tasks and jobs clean up after themselves. There is just one such condition: EXIT.

Debug conditions are provided to make the checkout or debugging of SCL procedures, etc. easier to accomplish. There is only one such condition: EXECUTION_FAULT.

The conditions for which a handler may be established are described in detail the subsection on *Condition Descriptions*.

11.2.2 Residence of Condition Handlers

Condition handlers can reside in the following kinds of SCL blocks.

JOB	This is the current block once a job has begun execution.
PROCEDURE or FUNCTION	A procedure block is established for the execution of an SCL command or function procedure.
TASK / TASKEND	Tasks initiated via the TASK/TASKEND statement, whether synchronous or asynchronous, may have condition handlers.
UTILITY	A utility block is established by a command utility. Actually, condition handlers within a utility reside in an input block associated with that utility. Note, that there can be more than one such block active for a utility at a time (e.g. from nested "include" commands/requests). Also note that this means that a condition handler established within the prolog for a utility, is automatically disestablished (cancelled) when the prolog completes, i.e. it is no longer established when the main input for the utility is processed.
WHEN / WHENEND	A condition handler may establish its own condition handlers.

11.2.3 WHEN / WHENEND

The WHEN/WHENEND statement establishes a condition handler for one or more conditions.

```
<when statement> ::= WHEN <sp> <condition name expression
                          [<,|sp> <condition name expression>]... <do>
                      [<statement list>]
                    WHENEND
```

A condition name expression specifies one of the names described in the *Condition Descriptions* section. If a handler is already established in the same block for a specified condition name, the new condition handler replaces the existing one for that condition name.

When one of the conditions for which the condition handler was established occurs, the condition handler is activated. This activation consists of two steps: creating the environment for the condition handler, and then processing the statement list of the condition handler.

The environment of the condition handler consists of a WHEN/WHENEND block statically linked to the block in which the handler was established, and within that block the variables described below, each of which is defined local to the WHEN/WHENEND block for the condition handler. These variables are:

OSV$CONDITION	This name variable identifies the type of condition that caused the condition handler to be activated.
OSV$LIMIT_NAME	This name variable is created only if OSV$CONDITION is LIMIT_FAULT. It identifies the particular limit that was exceeded.
OSV$STATUS	This status variable identifies the status for a fault or debug condition. It is normal for an informative condition. For an exit condition, it contains the completion status of a command or function procedure and is normal when exiting a job task or utility.
OSV$COMMAND	This string variable contains the image of the command that was active when the condition occurred. If the condition handler terminates its processing with the RETRY option of the CONTINUE statement, the command that is retried is that contained in this variable—as possibly modified by the condition handler.
OSV$COMMAND_NAME	This string variable is defined for compatibility with previous system levels.

It contains the name of the command that was active when the condition occurred.

If the condition for which the handler was activated occurs within the handler, and that condition is a fault condition, the condition handler is aborted.and processing continues as if the handler had not been established.

Note, that the command search mode used while processing a WHEN/WHENEND block may be different than the search mode in effect when the condition handler was established. This can happen to a handler established by a user if the condition occurs while a an application whcih has made the command search mode more restrictive is running.

11.2.4 CANCEL

The CANCEL statement disestablishes the condition handler(s) established for the specified conditions in the current block.

```
<cancel> ::= CANCEL <sp> <condition name expression>
                    [<,|sp> <condition name expression>]...
```

A condition name expression specifies one of the names described in the *Condition Descriptions* section. If no handler is established in the current block for a specified condition name, the CANCEL is ignored for that condition name.

11.2.5 CAUSE

The CAUSE statement is used to force a condition to occur. This is the only way to make a “user defined” condition occur.

```
<cause> ::= CAUSE <sp> <cause condition>

<cause condition> ::= CONDITION <sp> <condition name expression>
                        [<sp> WITH <sp> <status expression>]
                    | <sp> <status expression>
```

The value of the status expression is assigned to the OSV$STATUS variable in the condition handler for the condition being caused.

A condition name expression specifies one of the names described in the *Condition Descriptions* section. If the CONDITION clause is omitted, COMMAND_FAULT is assumed. The condition names EXIT, ANY_FAULT, or ANY_CONDITION may not be specified.

11.2.6 CONTINUE

The CONTINUE statement is used to exit from a condition handler or to pass the condition for which the handler was activated to another handler.

```
<continue> ::= CONTINUE [<sp> <continue option>] [<sp> <when clause>]

<continue option> ::= NEXT
                    | RETRY
                    | NEXT_HANDLER
                    | NEXT_USER_HANDLER

<when clause> ::= WHEN <boolean expression>
```

The NEXT option, which is the default, causes processing to continue with the thing that would have happened if the condition had not occurred.

The RETRY option is meaningful only for COMMAND_FAULT and EXECUTION_FAULT conditions and causes the statement that activated the condition handler to be retried. The handler may have modified the image of the command to be

retried via the OSV$COMMAND variable. For other conditions, RETRY is treated the same as NEXT.

The NEXT_HANDLER option causes the next condition handler established by a containing block to be invoked. If no such handler is found, the system's default handler for the condition is called. Depending on the condition and the action taken by the condition handler(s) to which control is passed using this option, control may be returned to the statement following the CONTINUE.

The NEXT_USER_HANDLER option is the same as the NEXT_HANDLER option except that the system's default handler for the condition is not called.

For the EXIT condition, the NEXT_HANDLER and NEXT_USER_HANDLER options are equivalent to the NEXT option. For the COMMAND_FAULT and EXECUTION_FAULT conditions, the NEXT_HANDLER and NEXT_USER_HANDLER options have the effect of causing processing to continue as if no condition handler had been established for the condition.

The CONTINUE is inhibited if the WHEN clause is present and its boolean expression value is FALSE.

11.2.7 Condition Descriptions

11.2.7.1 ANY_CONDITION

ANY_CONDITION can be used as a convenient way of specifying a handler for any fault or informative condition. Note that this does not include debug conditions, i.e. EXECUTION_FAULT, and exit conditons, i.e. EXIT.

11.2.7.2 ANY_FAULT

ANY_FAULT can be used as a convenient way of specifying a handler for any fault condition. Note that this does not include debug conditions, i.e. EXECUTION_FAULT, and exit conditons, i.e. EXIT.

11.2.7.3 COMMAND_FAULT

A COMMAND_FAULT occurs when an error is detected on the call to a statement or command for which no status parameter was supplied. A handler established for the COMMAND_FAULT condition will only be invoked if such an error occurs on a statement executed directly from the block that established the handler or a block statically contained within it (i.e. “included” via a reference to $COMMAND). If a COMMAND_FAULT is detected in a procedure or input block that does not have a handler established for the condition, the condition is turned into an EXECUTION_FAULT condition.

11.2.7.4 DISCONNECT

A DISCONNECT condition occurs when either the connection between the interactive user's terminal and the job is inadvertantly broken (line disconnect), or the user detaches the job.

11.2.7.5 EXECUTION_FAULT

An EXECUTION_FAULT can be thought of as an indirect COMMAND_FAULT, i.e. one that occurred in a block called indirectly by the block that established the EXECUTION_FAULT handler.

This condition is in the class of *debug* conditions, i.e. it is intended to aid in the checkout and debugging of procedures, etc. A handler should *not* be established for this condition once the checkout phase of development has been completed. The processing of EXECUTION_FAULT conditions introduces what may be an unacceptable amount for procedures and included files in normal use.

11.2.7.6 EXIT

An EXIT condition occurs when the block in which the condition handler resides is being terminated (exited) for *any* reason.

11.2.7.7 LIMIT_FAULT

A LIMIT_FAULT occurs when the amount of some processing resource, such as the amount of time allowed a job, has been reached.

11.2.7.8 PAUSE

A PAUSE condition occurs when the interactive user enters the pause break sequence.

11.2.7.9 RECONNECT

A RECONNECT condition occurs when the connection between the interactive user's terminal and the job is re-established after a DISCONNECT.

11.2.7.10 TERMINATE

A TERMINATE condition occurs when the interactive user enters the terminate break sequence.

11.2.7.11 UNSEEN_MAIL

An UNSEEN_MAIL condition occurs when, via Mail/VE, a “letter” is sent to a mailbox owned by the user on whose behalf the job is running.

11.2.7.12 Other Conditions

NOS/VE defines a number of other conditions for various purposes, mostly related to job recovery and file access. The names of these conditions all begin with OSC$, e.g. OSC$JOB_RECOVERY.

11.2.7.13 User Defined Conditions

Users may define their own conditions. A user defined condition occurs via a CAUSE statement that specifies a condition name other than one of those described in the preceding subsections.

12 Control Statements

12.1 CYCLE

The cycle statement causes the next iteration, if any, of a repetitive statement to be executed. In the case of a LOOP statement, this means that the first statement of the LOOP statement's statement list is given control. In the case of a WHILE or REPEAT statement, this means that the boolean expression controlling the loop is evaluated. In the case of a FOR statement, this means that the processing done at the FOR statement's FOREND clause is performed.

```
<cycle> ::= CYCLE [<sp> <label>] [<sp> <when clause>]

<when clause> ::= WHEN <sp> <boolean expression>
```

If a label is present in the CYCLE statement the enclosing repetitive statement with that label is cycled, otherwise the innermost repetitive statement is cycled.

The cycle is inhibited if the WHEN clause is present and its boolean expression value is FALSE.

12.2 EXIT

The EXIT statement causes a transfer of control out of a structured statement, parameter checking statement, utility, command procedure or function procedure.

```
<exit> ::= EXIT [<sp> <exit designator>] [<sp> <exit clause>]...

<exit clause := <when clause>
              | <exit with clause>]

<exit designator> ::= <label>
                    | <utility name> | UTILITY
                    | <command procedure name> | PROCEDURE
                    | <function procedure name> | FUNCTION
                    | TASK
                    | CHECK (*)

<exit with clause> ::= [<exit abort clause> <sp>] WITH <sp> <expression>

<exit abort clause> ::= ABORT
```

If an <exit designator> is present in the EXIT statement the enclosing structured statement, parameter checking statement, utility or procedure is exited; otherwise the innermost structured statement, parameter checking statement, utility or procedure is exited. Only a statically enclosing structured statement, the current task, or a statically enclosing parameter checking statement may be exited, however any active utility or procedure may be exited. UTILITY may be used to designate the innermost active utility. PROCEDURE may be used to designate the statically enclosing command procedure. FUNCTION may be used to designate the statically enclosing function procedure. If a procedure name is used it must be the first in the list of <command names> or <function names> in the procedure header. TASK may be used to designate the currently executing task. CHECK may be used to designate the statically enclosing CHECK/CHECKEND statement.

The WITH clause may not be used when exiting a structured statement or utility and must be used when exiting a function procedure or task. In the case of a function procedure, the value of the expression in the WITH clause becomes the result value for the function being exited. For a WITH clause specified when exiting a task, command procedure or parameter checking statement (*), the expression must evaluate to a status value and that status becomes the corresponding termination status. If the WITH clause is omitted when exiting a command procedure, termination with normal status occurs.

The ABORT clause may only be used when the <exit designator> is TASK. If the ABORT clause is specified in an EXIT TASK statement, and there is an “abort file” associated with the execution of the task, the DEBUG utility is invoked to process the abort file prior to the task's terminatioon.

The EXIT is inhibited if the WHEN clause is present and its boolean expression value is FALSE.

12.2.1 Exiting a Command Procedure

The EXIT statement can be used to leave a command procedure early, i.e. prior to getting to its PROCEND statement, or to terminate the command procedure with an abnormal status.

The EXIT statement is the only means to explicitly establish a command procedure's completion status. If an EXIT statement is not used or if its WITH clause is omitted, the command procedure terminates with normal status. The WITH clause of the EXIT statement can be used to specify a particular termination status. A command procedure should not explicitly write into its STATUS parameter. This will ultimately have no effect since the SCL interpreter will automatically update the status parameter as appropriate upon termination of the procedure. Also, if the STATUS parameter was omitted from the call to the procedure, attempting to write into it will result in an error.

12.2.2 Exiting a Function Procedure

The EXIT statement is the means of specifying the result to be returned by a function procedure. This means that a function procedure must perform an EXIT statement that has a WITH clause in order to complete properly.

12.2.3 Exiting a Utility

There are two variants to EXITing a command utility. The first appears similar to EXITing a structured statement. For example, the following sequence shows a conditional exit from the “source code utility”:

```
SOURCE_CODE_UTILITY
  .
  .
 EXIT UTILITY WHEN some_condition
  .
  .
QUIT
```

In this example, if *some_condition* is true, the statements following the EXIT statement up to and including the QUIT command are skipped, and normal processing resumes with the statement following QUIT.

The EXIT statement also provides the means for implementing a “utility termination command” for a utility implemented at the “command level,” i.e. via the UTILITY/UTILITYEND statement. For instance, the following is an example of an SCL procedure that implements the command used to “quit” a *hypothetical_utility*.

```
PROCEDURE quit, qui (
  status)

 EXIT hypothetical_utility

PROCEND quit
```

When the EXIT statement is used for this latter purpose, it is always best to explicitly name the utility to be exited, i.e. use a <utility name> for the <exit designator> rather than the generic UTILITY designator. This makes the EXIT statement work as expected when calls to utilities have been nested.

12.2.4 Exiting a Task

When the <exit designator> of an EXIT statement is TASK, the current task (executing program) is terminated with the status specified via the WITH clause. If the ABORT clause is omitted, the task is terminated by calling the PMP$EXIT program interface. If the ABORT clause is specified, the task is terminated by calling the PMP$ABORT program interface, thereby allowing an “abort file” associated with the task, if any, to be processed.

This form of the EXIT statement is intended for use within a WHEN/WHENEND statement to cause termination of a

program. For example, the following condition handler for LIMIT_FAULT resets the job's CPU_TIME limit and aborts the task which caused the fault.

```
WHEN limit_fault DO
  change_job_limit name=cp_time warning_limit_limit=unlimited
  EXIT TASK ABORT WITH osv$status
WHENEND
```

13 Command List

The ability to manipulate the command list is primarily intended for use by installation management personnel and utility writers to establish a user repertoire by adding entries to or deleting entries from the SCL command list. Normally, users need not be concerned with the SCL command list. The system automatically defines a command repertoire as a part of the user validation and job initialization process.

The command list is used to search for a number of things in addition to commands. These other things are related to command processing and include functions, message templates, prompting and help information for commands entered interactively, and screen formatting “panels” (forms).

13.1 Command List Entries

A command list entry may be any one of the items described in the following subsections.

13.1.1 Object Library

A file used as a command list entry must be an object library. An object library can contain commands, functions, message modules, help modules, and screen formatting panels, in addition to executable programs. An object library intended for use in the command list is often called a command library. An object library containing only SCL procedures is sometimes called a procedure library.

To be added to a user's command list, the user must have at least EXECUTE permission to the library. Also, the library must be added to the command list from a ring that is less than or equal to the R3 component of its RING_ATTRIBUTES.

A library in the command list cannot be overwritten or released from the job.

For a command to be found in an object library, the command name must be one of the names of a *command procedure*, one of the names of a *program description*, one of the names of a *command description*, or the name of the starting procedure in an object module. In addition, to be called from a file other than the object library itself the command must have the XDCL or GATE scope attribute.

For a function to be found in an object library, the function name must be one of the names of an *function procedure* or one of the names of a *function description*. In addition, to be called from a file other than the object library itself, the function must have the XDCL or GATE scope attribute.

For a message template to be found in an object library, the status condition code corresponding to the message template must be defined in one of the *message modules* on the library.

Help modules only exist on object libraries and are used when processing a command or function interactively, and by some applications. Help modules are searched for by name.

For a screen formatting panel (form) to be found in an object library, the form name must be the name of a *panel module*.

13.1.2 Catalog

As far as the command list is concerned, a catalog can contain only commands.

For a command name to be found in a catalog entry, the command name must be the name of a file in the catalog and the file must contain an executable program or an SCL command procedure. The FILE_CONTENTS attribute of the file is used to determine how to process the command.

When FILE_CONTENTS is OBJECT_DATA or OBJECT_LIBRARY the program is executed.

When FILE_CONTENTS is LEGIBLE_SCL_PROCEDURE, LEGIBLE_DATA or UNKNOWN, the SCL command procedure is interpreted. The file containing the procedure may not have a user defined file access procedure

(FAP) associated with it and it may not have the attribute combination RECORD_TYPE of UNDEFINED (U) and BLOCK_TYPE of SYSTEM_SPECIFIED (SS).

13.1.3 Working Catalog

The working catalog can be in the command list generically. When this command list entry is to be searched, the current value of the WORKING_CATALOG environment object is determined, and that catalog is then processed as described in the preceding section.

13.1.4 $system

$SYSTEM represents the command list entry containing all of the commands available to a user that are provided by NOS/VE.

The $SYSTEM command list entry contains the commands that are in the “control statements” and “control commands” entries. This allows those commands to be called in the “$system.command_name” form.

(The $SYSTEM command list entry actually represents two partial entries that are perceived by the user as a single unified entry. The first part is a set of commands, functions, message and help modules that are built in to the system. The second part is an object library, the “system command library” which is normally $SYSTEM.OSF$COMMAND_LIBRARY. When the $SYSTEM command list entry is searched, the system command library is searched first, then the built in part is searched.)

13.1.5 Command Utility

The subcommands and functions provided by a command utility constitute a command list entry.

The commands and functions of a command utility established at the program interface level, via CLP$BEGIN_UTILITY, are callable only within the task in which the utility was established. Such commands and functions are, however, “visible” outside the establishing task, e.g. via DISPLAY_COMMAND_LIST_ENTRY.

13.1.6 Control Statements

All SCL statements that are not commands belong to a special command list entry that is **always** the first entry in the command list.

The following commands also belong to this group: COLLECT_TEXT, JOB/JOBEND, UTILITY/UTILITYEND and TASK/TASKEND. This group cannot be removed from or replaced in the command list.

13.1.7 Control Commands

There are a few commands that provide capabilities basic to SCL and belong to a special command list entry that is *always* the last entry in the command list.

The commands in this group are:

```
CHANGE_COMMAND_SEARCH_MODE
CHANGE_SYSTEM_COMMAND_LIBRARY
CREATE_COMMAND_LIST_ENTRY
CREATE_VARIABLE_LIST_ENTRY (*)
DELETE_COMMAND_LIST_ENTRY
DELETE_VARIABLE_LIST_ENTRY (*)
DELETE_VARIABLE
DISPLAY_COMMAND_ENVIRONMENT (*)
DISPLAY_COMMAND_INFORMATION
DISPLAY_COMMAND_LIST
DISPLAY_COMMAND_LIST_ENTRY
DISPLAY_FUNCTION_INFORMATION
DISPLAY_VALUE
DISPLAY_VARIABLE_LIST
DISPLAY_VARIABLE_LIST_ENTRY (*)
DISPLAY_WORKING_CATALOG
```

```
GET_LINE
INCLUDE_COMMAND
INCLUDE_FILE
INCLUDE_LINE
LOGIN
LOGOUT
PUT_LINE
WAIT
```

13.1.8 Fence

A FENCE entry serves as a marker in the command list and is used in conjunction with certain modes of searching for commands and functions. See the discussion of searching the command list, below.

13.2 Command List Searching

Which entries of the command list are searched for commands and functions can be controlled by the presence of a “fence” in the command list and the search mode. These search constraints only apply to searches for commands and functions; the entire command list is searched for message templates, help modules, and screen formatting panels (forms).

Normally all entries in the command list are searched. A command utility, a site, or a user can restrict command and function searches to certain entries, while still permitting callable applications to have access to the entire command list. An “application,” in this sense, is the processor (procedure or program) for a command or function.

13.2.1 Fences

A “fence” in the command list separates those commands and functions that are directly callable by users from those that can be called only by an application.

A fence is defined in one of the following ways:

1. The special FENCE entry is added to the command list by specifying its keyword on the CREATE_COMMAND_LIST_ENTRY command.
2. A command utility is initiated which makes the search mode more restricted, i.e. if the current search mode is GLOBAL, setting it to RESTRICTED or EXCLUSIVE, or if the current search mode is RESTRICTED, setting it to EXCLUSIVE. In this case the fence is assumed to immediately follow the utility's command list entry.
3. If a fence has not been defined by either of the first two methods, a fence is assumed to immediately follow the first entry in the command list (not counting the special “control statements” entry).

Although it possible to have a two, or even three, fences in the command list at a time, as defined by the first two methods, only the fence closest to the beginning of the command list is used.

A fence is used in the following situations:

1. A command is called by just specifying its name and the search mode is RESTRICTED or EXCLUSIVE. In this case only those command list entries that precede the fence are searched for the command. Entries following the fence are not searched.
2. A function is called when the search mode is EXCLUSIVE. In this case only those command list entries that precede the fence are searched for the function. Entries following the fence are not searched.
3. A command is called by preceding its name with a slant (/) character and the search mode is RESTRICTED or GLOBAL. In this case the search begins with the entry following the fence. Entries that precede the fence are not searched.

13.2.2 Search Modes

The three ways a command list can be searched are:

GLOBAL	All entries in the command list are candidates for being searched. Also, commands which are specified by file name and command name can be called, and the slant (/) character may be used to cause the search to begin after the fence.
RESTRICTED	All entries in the command list are candidates for being searched but in order to use a command that is not in an entry which comes before the fence, the command must be preceded by a slant (/) character. Also, commands which are specified by file name and command name can be called.
EXCLUSIVE	Only the entries that precede the fence are candidates for being searched. In this mode, commands which are specified by file name and command name are not allowed, nor may a command be preceded by the slant (/) character in an attempt to skip past the fence. Also, the DISPLAY_COMMAND_LIST command does not display those entries which follow the fence. (This also applies to the ALL display option of the DISPLAY_COMMAND_LIST_ENTRY command.)

In both RESTRICTED and EXCLUSIVE search mode, the following commands are not allowed:

CHANGE_COMMAND_SEARCH_MODE
CHANGE_SYSTEM_COMMAND_LIBRARY
CREATE_COMMAND_LIST_ENTRY
DELETE_COMMAND_LIST_ENTRY

Regardless of the search mode the “control statements” and “control commands” entries are **always** candidates for being searched for a command; and the $SYSTEM entry is **always** a candidates for being searched for a function (it is searched last if it is not one of the entries already searched).

13.2.3 Effective Search Mode

The search modes described above apply to the search for commands and functions called directly by users. Although it may be desirable for security or other reasons to restrict what a user can do, the applications which a user is allowed to call must have access to the full command repertoire. Therefore, within an application—command or function processor—the search mode is always considered to be GLOBAL.

This gives rise to the concept of an *effective search mode*, i.e. the search mode that is actually used as opposed to the search mode selected via the CHANGE_COMMAND_SEARCH_MODE command or by a command utility when it starts up. The effective search mode is the selected mode for commands and functions called by a user, and GLOBAL for commands and functions called by an application. The effective search mode applies only to searches, not to displays.

(The effective search mode is also GLOBAL when executing at or below ring 3. This fact should only be of interest within the operating system.)

13.3 Commands Related to the Command List

13.3.1 change_command_search_mode (chacsm)

This command changes the command search mode. It cannot be used when the search mode is EXCLUSIVE, or if a command utility is active and the search mode is RESTRICTED.

Besides changing the selected search mode for the command list, this command also changes *effective* search mode of its immediate caller. For example, if a command procedure uses this command to change the search mode to EXCLUSIVE, that change will apply to all remaining commands and functions called by the procedure.

```
change_command_search_mode, chacsm (
      search_mode, sm: key
          (global, g)
          (restricted, r)
          (exclusive, e)
        keyend = $required
      status)
```

SEARCH_MODE, SM: This parameter specifies the new search mode to be associated with the command list. See the section on command list search modes for a description of these options.

STATUS: See the section on Error Handling.

13.3.2 change_system_command_library (chascl)

This “advanced usage” command provides a way to change the object library associated with the $SYSTEM command list entry. It cannot be used when the command list search mode is EXCLUSIVE or RESTRICTED.

```
change_system_command_library, chascl (
      system_command_library, scl: any of
          key
            (standard, s)
          hidden_key
            none
          keyend
          file
        anyend = $required
      status)
```

SYSTEM_COMMAND_LIBRARY, SCL: This parameter specifies which library should be associated with the $SYSTEM command list entry. The STANDARD keyword selects $SYSTEM.OSF$COMMAND_LIBRARY. (The NONE keyword allows there to be no library associated with the $SYSTEM command list entry. This capability is probably only of use to the SCL project, which is why the keyword is hidden.)

STATUS: See the section on Error Handling.

13.3.3 create_command_list_entry(ies) (crecle)

This command adds entries to either the beginning or end of the command list. It cannot be used when the command list search mode is EXCLUSIVE or RESTRICTED.

```
create_command_list_entry, create_command_list_entries, crecle (
      entry, entries, e: list of any of
          key
            $system
          advanced_key
            fence
          keyend
          file
        anyend = $required
      placement, p: key
          (after, a)
          (before, b)
        keyend = before
      status)
```

ENTRY, ENTRIES, E: This parameter specifies entries to be added to the command list. They will appear in the command list in the order specified. Note that if `:$WORKING_CATALOG` is added as a command list entry, it always represents the current working catalog, not necessarily what the working catalog was when the entry was added. (The leading colon is crucial for this distinction. See the section on *File Expressions* for details on its significance.)

PLACEMENT, P This parameter specifies whether the ENTRIES added to the command list are placed BEFORE or AFTER the current entries.

Omission will cause BEFORE to be used.

STATUS: See the section on Error Handling.

13.3.4 delete_command_list_entry(ies) (delcle)

This command deletes entries from the command list. It cannot be used when the command list search mode is EXCLUSIVE or RESTRICTED.

```
delete_command_list_entry, delete_command_list_entries, delcle (
      entry, entries, e: any of
          key
            all
          keyend
          list of any of
            key
              $system
            advanced_key
              fence
            keyend
            file
          anyend
        anyend = $required
      status)
```

ENTRY, ENTRIES, E: This parameter specifies the entries to be deleted from the command list. If this parameter is specified as ALL, then all entries in the command list are deleted. Utilities cannot be deleted from the command list with this command.

STATUS: See the section on Error Handling.

13.3.5 display_command_list (discl)

This command displays the names of the entries in and/or the search mode governing the command list.

```
display_command_list, discl (
      display_options, display_options, do: list of key
          all
          (entries, entry, e)
          (search_mode, sm)
        keyend = entries
      output, o: file = $output
      status)
```

DISPLAY_OPTIONS, DISPLAY_OPTION, DO: This parameter selects the information to be displayed.

ALL: This option selects all of the other options.

ENTRIES, ENTRY, E: This option causes the names of the entries in the command list to be displayed. If the search mode is EXCLUSIVE entries that follow the fence are not displayed.

SEARCH_MODE, SM: This option causes the command list search mode to be displayed.

Omission causes ENTRIES to be selected.

OUTPUT, O: This parameter designates the file to which the information is written.

Omission causes $output to be used.

STATUS: See the section on Error Handling.

13.3.6 display_command_list_entry(ies) (discle)

This command displays information about one or more command list entries.

```
display_command_list_entry, display_command_list_entries, discle (
```

```
      entry, entries, e: list of any of
          key
            all
            (control_statements, control_statement, cs)
            (first, f)
            $system
          keyend
          file
        anyend = first
      display_options, display_options, do: list of key
          all
          (all_names, all_name, an)
          (commands, command, c)
          (functions, function, f)
          (names, name, n)
          (advanced_usage, au)
          (starting_procedures, starting_procedure, sp)
        keyend = osd$discle_display_options, (commands, names)
      output, o: file = $output
      status)
```

ENTRY, ENTRIES, E: This parameter selects the command list entry or entries to be displayed. Any entry that can be put in the command list can be selected regardless of whether it is presently in the command list. In addition the following keywords can be used to make generic selections.

ALL: This option selects the entire command list for display (this includes the control statements).

CONTROL_STATEMENTS, CONTROL_STATEMENT, CS: This option selects the control statements and commands inherently part of SCL.

FIRST, F: This option selects the first entry in the command list. This provides a way of selecting a display of the subcommands supplied by a utility when the utility is active.

$SYSTEM: This option selects the special command list entry that contains all of the system supplied command and functions.

Omission causes FIRST to be used.

DISPLAY_OPTIONS, DISPLAY_OPTION, DO: This parameter selects the form and content of the output.

ALL: This option selects all of the other options.

ALL_NAMES, ALL_NAME, AN: This option causes all of the names (abbreviations and aliases, as well as the “nominal” names) of commands and/or functions and/or control statements to be displayed.

COMMANDS, COMMAND, C: This option causes information about the individual commands within the selected command list entries (other than catalogs) to be displayed.

FUNCTIONS, FUNCTION, F: This option causes information about the individual functions within the selected command list entries to be displayed.

NAMES, NAME, N: This option causes the “nominal” names of commands and/or functions and/or control statements to be displayed. A “nominal” name is the name by which the command, function or statement is normally known.

ADVANCED_USAGE, AU: This option causes those commands and/or functions designated as being for “advanced usage” to be displayed.

STARTING_PROCEDURES, STARTING_PROCEDURE, SP: This option causes those commands whose names are starting procedures (entry points) in object modules to be displayed. These are not displayed by default on the theory that usually a program description or command description will be used to access to the program as a command.

Omission of this parameter causes COMMANDS and NAMES to be selected. Omission of both COMMANDS and FUNCTIONS causes COMMANDS to be selected. The default can be overridden by creating a “default variable” called OSD$DISCLE_DISPLAY_OPTIONS and putting a string representing your favorite options into it.

OUTPUT, O: This parameter designates the file to which the information is written.

Omission causes $output to be used.

STATUS: See the section on Error Handling.

13.4 Functions Related to the Command List

13.4.1 $command_list

The $COMMAND_LIST function returns the user definable portion of the command list. It is not affected by the selected search mode as is the DISPLAY_COMMAND_LIST command. It has no parameters.

```
$command_list (
      )
```

Each element of the returned list is either of type file (for object library and catalog entries), type name (for command utility entries or type keyword for the $SYSTEM and FENCE entries).

13.4.2 $command_list_entry

The $COMMAND_LIST_ENTRY function returns a list of the commands or functions in a command list entry.

```
$command_list_entry (
      entry: any of
          key
            (first, f)
            $system
          keyend
          name
          file
        anyend = first
      commands_or_functions: key
          (commands, command, c)
          (command_references, command_reference, cr)
          (functions, function, f)
        keyend = commands
      options: list rest of key
          (aliases, abbreviation, alias, a)
          (advanced_usage, au)
        keyend = $optional
      )
```

ENTRY: This parameter selects the command list entry for which information is to be returned. Any entry that can be put in the command list can be selected regardless of whether it is presently in the command list. In addition the following keywords can be used to make generic selections.

FIRST selects the first entry in the command list. This provides a way of selecting a display of the subcommands supplied by a utility when the utility is active.

$SYSTEM selects the special command list entry that contains all of the system supplied command and functions.

Omission causes FIRST to be used.

COMMAND_OR_FUNCTIONS: This parameter selects whether information about commands or functions is to be returned.

COMMANDS causes the names of commands to be returned.

COMMAND_REFERENCES causes command_references for the commands to be returned.

FUNCTIONS causes the names of functions to be returned.

Omission causes COMMANDS to be used.

OPTIONS: This parameter selects which commands or functions within ENTRY are to be returned.

ALIASES causes a “list of list of element” to be returned where each sublist contains elements for all the names of a particular command or function. If ALIASES is not selected, a “list of element” is returned where each element corresponds to the “nominal” name of a command or function. (The term “element” refers to either name or

command_reference depending on which was selected.)

ADVANCED_USAGE causes commands or functions with this attribute to be included in the result, otherwise only NORMAL_USAGE commands or functions are returned.

13.4.3 $command_search_mode

The $COMMAND_SEARCH_MODE function returns a keyword which represents the current command search mode. It has no parameters.

```
$command_search_mode (
      )
```

The keyword returned is one of GLOBAL, RESTRICTED or EXCLUSIVE.

13.4.4 $source

The $SOURCE function is used to determine where the processor for the requesting command was found. The returned value is of type file if the source of the command is a file or catalog, the name of the utility if the command is a subcommand of a utility, or the name $SYSTEM if the command is a system supplied command. It has no parameters.

```
$source (
      )
```

When used in the expression for a parameter to a command this function will return the source of that command. In such circumstances, the $source_of_caller function is often more appropriate.

13.4.5 $source_of_caller

The $SOURCE_OF_CALLER function is used to determine where the processor for the command that called the requesting command was found. The returned value is of type file if the source of the command's caller is a file or catalog, the name of the utility if the command's caller is a subcommand of a utility, or the name $SYSTEM if the command's caller is a system supplied command. It has no parameters.

```
$source_of_caller (
      )
```

14 Command Utilities

A command utility provides an environment in which to perform a particular set of operations. Most important to this environment is a set of commands, known as the subcommands of the utility, through which the operations are actually performed. In addition, command utilities often provide functions that ease the job of “programming” the utility.

One of the most attractive features of the environment created by a command utility is that all of the system or user supplied commands are still available while the utility is active. A command utility augments, rather than replaces, the environment that existed when the utility was called.

A command utility is defined and executed by use of the UTILITY/UTILITYEND command.

14.1 UTILITY / UTILITYEND

The purpose of this command is to delimit the definition and execution of a command utility. The parameters of this command provide the definition of the utility environment. Statements and commands between the UTILITY command and UTILITYEND delimiter specify the execution of the utility. At a minimum this statement list should contain an INCLUDE_FILE or INCLUDE_LINE command to specify the commands to be executed within the context of the established utility environment.

The commands and functions that are provided by the utility are defined by a set of COMMAND and FUNCTION “commands” provided on a file specified by the TABLES parameter. If the TABLES file is $COMMAND (or $COMMAND_OF_CALLER), a TABLEND “command” must be used to designate the end of the definition of the command and function tables and the beginning of the statements to be executed within the established utility.

```
utility (
      name, n: name = $required
      enable_subcommand_logging, esl: (by_name) boolean = yes
      interactive_include_processor, iip: (by_name) any of
          key
            none
          keyend
          name
        anyend = none
      library, libraries, l: (by_name) list of file
      line_preprocessor, lp: (by_name) any of
          key
            none
          keyend
          name
        anyend = none
      online_manual, om: (by_name) any of
          key
            none
          keyend
          name
        anyend = none
      prompt, p: (by_name) string 0 .. 27
      search_mode, sm: (by_name) key
          (global, g)
          (restricted, r)
          (exclusive, e)
        keyend = global
      tables, table, t: (by_name) file = $command
      termination_command_name, tcn: (by_name) name = quit
      status)
```

NAME, N: This parameter specifies the name of the utility. This name is used to refer to the utility while it is active.

ENABLE_SUBCOMMAND_LOGGING, ESL: This parameter determines whether the utility's subcommands may be logged (YES) or not (NO). Logging of commands occurs only for commands read from the main command file

(:$LOCAL.COMMAND) of a job. If this parameter is set to NO, the subcommands belonging to the utility are not logged even if they are read from this file. Commands not belonging to the utility are unaffected by this parameter.

Omission causes YES to be used.

INTERACTIVE_INCLUDE_PROCESSOR, IIP: (*) This parameter specifies the name of the command to be called each time an INCLUDE_FILE command/request is issued on behalf of the utility that references an interactive file. See the discussion below of interactive include processors.

Omission causes no interactive include processor to be used.

LIBRARY, LIBRARIES, L: This parameter specifies the library or libraries on which the processors for the commands and functions can be found.

Omission causes the library containing the call to the UTILITY/UTILITYEND command to be used.

LINE_PREPROCESSOR, LP: (*) This parameter specifies the name of the command to be called each time a command line is read prior to interpreting the line. See the discussion below of line preprocessors.

Omission causes no line preprocessor to be used.

ONLINE_MANUAL, OM: This parameter specifies the name of the online manual that describes the utility.

Omission causes no online manual to be associated with the utility.

PROMPT, P: This parameter specifies the prompt string to be used for interactive command input. For command lines, a "/" character is appended to the specified prompt string. For command continuation lines, the string "../" is appended to the specified prompt string.

Omission causes the utility name to be used.

SEARCH_MODE, SM: This parameter specifies the command search mode to be used while processing commands for the uility. Strictly speaking, the search mode is an attribute of the command list, not of a utility; but is provided on this command because of its effect on the searching for utility supplied commands and commands from other command list entries. See the section on command list search modes for a description of these options. Upon exit from the utility, the command search mode is restored to the value it had when the utility was entered.

Omission causes GLOBAL to be used.

TABLES, TABLE, T: This parameter specifies the file containing the COMMAND and FUNCTION "commands" that define the tables to be searched for commands and functions once the utility environment is established.

Omission causes $COMMAND (i.e. the file containing the UTILITY/UTILITYEND command) to be used.

TERMINATION_COMMAND_NAME, TCN: This parameter specifies the utility subcommand that is used to terminate the utility. This must be one of the commands that was defined by a COMMAND command in the file specified by the TABLES parameter.

Omission causes QUIT to be assumed as the name of the termination command for the utility.

STATUS: See the section on Error Handling.

The following subsections describe the COMMAND, FUNCTION, and TABLEND "commands" read from the file specified by the TABLES parameter.

14.1.1 Command

The COMMAND command declares a group of entries in a command table. The order in which the names for the entries are provided has significance when the table is used by the system supplied DISPLAY_COMMAND_LIST_ENTRY command and in other similar contexts. The first entry name is taken to be the "nominal" name for the command, i.e., the name by which it is usually known. The last entry name is taken to be the "abbreviation" for the command name. Any other entry names are considered to be aliases for the command name.

```
command (
      name, names, n: list of name = $required
      processor, p: name
      availability, a: key
          (normal_usage, advertised, a, nu)
          (advanced_usage, advanced, au)
```

```
        (hidden, h)
      keyend = normal_usage
    automatically_log, al: boolean = yes
    )
```

NAME, NAMES, N: This parameter specifies the names by which the command can be called.

PROCESSOR, P: This parameter specifies the name of the processor for the command.

Omission causes the first element of the NAME parameter to be used.

AVAILABILITY, A: This parameter specifies whether the command is thought to be for NORMAL_USAGE (NU) or ADVANCED_USAGE (AU), or is to be HIDDEN (H) from users. An ADVANCED_USAGE command will only appear in a display of the command list if explicitly requested. A HIDDEN command will not appear in a display of the command list.

Omission causes NORMAL_USAGE to be used.

automatically_log, al: This parameter specifies whether tha SCL interpreter should log the command when it is recognized (YES), or leave the logging of the command to the command processor (NO). A command processor may choose to log calls itself in order to suppress secure information.

Omission causes YES to be used.

14.1.2 Function

The FUNCTION command declares a group of entries in a function table. The order in which the names for the entries are provided has significance when the table is used by the system supplied DISPLAY_COMMAND_LIST_ENTRY command and in other similar contexts. The first entry name is taken to be the “nominal” name for the function, i.e., the name by which it is usually known. The last entry name is taken to be the “abbreviation” for the function name. Any other entry names are considered to be aliases for the function name.

```
function (
      name, names, n: any of
          data_name
          list of data_name
        anyend = $required
      processor, p: data_name
      availability, a: key
          (normal_usage, advertised, a, nu)
          (advanced_usage, au)
          (hidden, h)
        keyend = advertised
      )
```

NAME, NAMES, N: This parameter specifies the names by which the function can be called.

PROCESSOR, P: This parameter specifies the name of the processor for the function.

Omission causes the first element of the NAME parameter to be used.

AVAILABILITY, A: This parameter specifies whether the function is thought to be for NORMAL_USAGE (NU) or ADVANCED_USAGE (AU), or is to be HIDDEN (H) from users. An ADVANCED_USAGE function will only appear in a display of the function list if explicitly requested. A HIDDEN function will not appear in a display of the function list.

Omission causes NORMAL_USAGE to be used.

14.1.3 TABLEND

The TABLEND command terminates the collection of definitions for a utility's command and/or function table(s). It has no parameters.

```
tablend (
```

```
)
```

14.2 Commands Related to Utilities

In addition to the commands described in the following subsections, the INCLUDE_FILE and INCLUDE_LINE commands play an important role with respect to command utilities in that one of them is usually the command used to initiate processing of the actual subcommands for a utility call.

14.2.1 change_utility_attribute(s) (chaua)

The CHANGE_UTILITY_ATTRIBUTES command changes attributes for a utility initiated by the UTILITY/UTILITYEND command.

```
change_utility_attributes, change_utility_attribute, chaua (
      utility, u: name = $required
      enable_subcommand_logging, esl: (by_name) boolean
      interactive_include_processor, iip: (by_name) any of
          key
            none
          keyend
          name
        anyend
      line_preprocessor, lp: (by_name) any of
          key
            none
          keyend
          name
        anyend
      online_manual, om: (by_name) any of
          key
            none
          keyend
          name
        anyend
      prompt, p: (by_name) string 0 .. 27
      tables, table, t: (by_name) file
      status)
```

UTILITY, U: This parameter specifies the name of the utility whose attributes are to be changed.

ENABLE_SUBCOMMAND_LOGGING, ESL: This parameter determines whether the utility's subcommands may be logged (YES) or not (NO). Logging of commands occurs only for commands read from the main command file (:$LOCAL.COMMAND) of a job. If this parameter is set to NO, the subcommands belonging to the utility are not logged even if they are read from this file. Commands not belonging to the utility are unaffected by this parameter.

If this parameter is omitted the subcommand logging enabled attribute is not changed.

INTERACTIVE_INCLUDE_PROCESSOR, IIP: (*) This parameter specifies the name of the command to be called each time an INCLUDE_FILE command/request is issued on behalf of the utility that references an interactive file. See the discussion below of interactive include processors.

If this parameter is omitted the interactive include processor attribute is not changed.

LINE_PREPROCESSOR, LP: (*) This parameter specifies the name of the command to be called each time a command line is read prior to interpreting the line. See the discussion below of line preprocessors.

If this parameter is omitted the line preprocessor attribute is not changed.

ONLINE_MANUAL, OM: This parameter specifies the name of the online manual that describes the utility.

If this parameter is omitted the online manual attribute is not changed.

PROMPT, P: This parameter specifies the prompt string to be used for interactive command input. For command lines, a “/” character is appended to the specified prompt string. For continuation lines, the stirng “../” is appended to the specified

prompt string.

If this parameter is omitted the prompt attribute is not changed.

TABLES, TABLE, T: This parameter specifies the file containing the COMMAND and FUNCTION “commands” that define the tables to be searched for commands and functions in the utility environment. If a COMMAND “command” is present in the file the current command table for the utility is replaced by the new table resulting from processing this file. If a FUNCTION “command” is present in the file the current function table for the utility is replaced by the new table resulting from processing this file.

If this parameter is omitted the command and functions tables for the utility are not changed.

STATUS: See the section on Error Handling.

14.3 Functions Related to Utilities

14.3.1 $utility

The $utility function is used to interogate attributes of the currently running command utility.

```
$utility (
      attribute: key
          (catalog, c)
          (name, n)
          (online_manual, om)
          (prompt, p)
          (subcommand_logging_enabled, scle, sle)
        keyend = catalog
      )
```

ATTRIBUTE: This parameter specifies the attribute to be returned.

CATALOG: (*) the name of the temporary catalog associated with the utility (if no utility is currently active, the value returned is equivalent to $LOCAL)

NAME: the name of the currently active command utility (if no utility is currently active, NONE is returned)

ONLINE_MANUAL: the name of the online manual associated with the utility (if no online manual is associated with the utility or no utility is currently active, NONE is returned)

PROMPT: the string that is being used by the utility to prompt for interactive command input (if no utility is currently active, the null string is returned)

SUBCOMMAND_LOGGING_ENABLED: a boolean value that answers the question: “is logging of utility subcommands enabled within the utility?” (if no utility is currently active, TRUE is returned)

Omission causes CATALOG to be used. Until this option is available, this parameter is $REQUIRED.

14.4 Interactive Include Processors (*)

A command utility may choose to be given control each time command processing begins from an interactive file on behalf of the utility.

This “hook” is provided for those utilities that provide a “screen interface,” for example: EDIT_FILE. The idea is that the interactive include processor would initiate the “screen interaction” with the user, if appropriate. When control is returned from the interactive include processor, further processing is determined as follows:

1. If the returned status is abnormal, the include file command or request terminates with that status.
2. If the utility's termination command has been executed, the include file command or request terminates with normal status.
3. Otherwise processing of commands from the interactive file takes place as if there had been no interactive include processor.

```
interactive_include_processor (
      interaction_style, is: (by_name) key
          (line, l)
          (screen, s)
          (desktop, dt, d)
        keyend
      status)
```

INTERACTION_STYLE, IS: This parameter specifies the user selected interaction style at the point at which the include file command or request was issued.

STATUS: See the section on Error Handling.

14.5 Command Line Preprocessors (*)

A command utility may choose to be given control each time a command line is read.

This “hook” is provided for those utilities whose input is not completely compatible with SCL. Such incompatibility may result from a need to meet an industry standard or retain compatibility with a predecessor product. Also, a utility may simply wish to be notified that a command line has been read; for example, a line boundary may be a convenient point to which to back up for a utility that supports an “undo” feature.

This capability allows a utility to use SCL facilities for reading commands when that would otherwise not be possible. Also, users of such a utility may still be able to make use of SCL's programming language facilities in conjunction with the utility.

The line preprocessor command must have the following parameters.

```
line_processor (
      command_line, cl: (by_name) string
      interactive_source, is: (by_name) boolean
      interpreting_commands, ic: (by_name) boolean
      edited_command_line, ecl: (var, by_name) string
      status)
```

COMMAND_LINE, CL: This parameter specifies the command line read by the SCL interpreter.

INTERACTIVE_SOURCE, IS: This parameter specfies whether the command line originated from an interactive terminal (TRUE) or not (FALSE).

INTERPRETING_COMMANDS, IC: This parameter specifies whether the SCL interpreter would interpret (YES) the next command or skip it (NO). Skip mode is entered as a result of various control statements, for example IF, ELSE, etc. If the utility processes the commands in the command line itself, it should only do so when this parameter is YES.

EDITED_COMMAND_LINE, ECL: This parameter specifies the string variable to which the preprocessed command line is written. Upon entry to the line preprocessor, the EDITED_COMMAND_LINE variable has the same value as the COMMAND_LINE parameter. The preprocessor may change (edit) the line in any manner it chooses. If the EDITED_COMMAND_LINE is set to the null (empty) string, the SCL interpreter will do no further processing with this line.

status: See the section on Error Handling.

14.6 An Example

This example shows the definition of a utility. This utility is comprised of a procedure to “get into” the utility environment and a procedure for each of the utility's subcommands. All of the procedures are assumed to reside on the same library.

```
          "---------- This is the main procedure.

          PROCEDURE date_time_utility, dtu (
```

```
  output, o: file = $output
  status)

  UTILITY name=date_time_utility prompt='dtu'
      command (change_format, chaf) processor=dtp#change_format
      command (display_date, disd) processor=dtp#display_date
      command (display_time, dist) processor=dtp#display_time
      command (quit, end, qui) processor=dtp#quit
    TABLEND

    VAR
      dtv$output: (utility) file = output
      dtv$date_format: (utility) string = 'DN MN D2, Y4'
      dtv$time_format: (utility) string = 'AMPM'
    VAREND

    include_file $command_of_caller utility=date_time_utility
  UTILITYEND

PROCEND date_time_utility


"---------- This is the procedure for change_format.

PROCEDURE dtp#change_format (
  date_format, df: string
  time_format, tf: string
  status)

  IF $specified(date_format) THEN
    dtv$date_format = date_format
  IFEND
  IF $specified(time_format) THEN
    dtv$time_format = time_format
  IFEND

PROCEND dtp#change_format


"---------- This is the procedure for display_date.

PROCEDURE dtp#display_date (
  format, f: string = dtv$date_format
  output, o: file = dtv$output
  status)

  display_value $date(format) o=output

PROCEND dtp#display_date


"---------- This is the procedure for display_time.

PROCEDURE dtp#display_time (
  format, f: string = dtv$time_format
  output, o: file = dtv$output
  status)

  display_value $time(format) o=output
```

```
PROCEND dtp#display_time

"---------- This is the procedure for quit.
PROCEDURE dtp#quit
  EXIT date_time_utility
PROCEND dtp#quit
```

15 format_scl_procedure (forsp)

The FORMAT_SCL_PROCEDURE command is a tool used to format a file of SCL commands.

The FILE_CONTENTS attributes of the input file, and of the output file if it exists, must be one of: LEGIBLE_SCL_PROCEDURE, LEGIBLE_SCL_INCLUDE, LEGIBLE_SCL_JOB, LEGIBLE_DATA or UNKNOWN.

If FORMAT_SCL_PROCEDURE creates the output file it will set its FILE_CONTENTS attribute to that of the input file, unless the input file's FILE_CONTENTS is UNKNOWN, in which case LEGIBLE_DATA is used. Other attributes will have the standard defaults.

```
format_scl_procedure, format_scl_procedures, format_scl_proc, format_scl_procs, ..
            forsclp, forsp (
      input, i: file = $required
      output, o: file = $required
      page_width, pw: (by_name) integer 65..65535 = 80
      initial_indent_column, iic: (by_name) integer 1..65535 = 1
      key_character, kc: (by_name) string 1 = '*'
      utility_definition_file, udf: (by_name) file
      process_collect_text, pct: (by_name) boolean = false
      translate, t: (by_name) boolean = false
      status)
```

INPUT, I: This parameter specifies the file from which the formatter is to read the SCL procedures or commands.

Omission causes $INPUT to be used.

OUTPUT, O: This parameter specifies the file to which the formatter is to write the formatted SCL procedures or commands.

Omission causes $OUTPUT to be used.

PAGE_WIDTH, PW: This parameter specifies the maximum length of a formatted line to be written to the OUTPUT file.

Omission causes 80 to be used. The value of the PAGE_WIDTH parameter must be at least 64 greater than the value of the INITIAL_INDENT_COLUMN parameter.

INITIAL_INDENT_COLUMN, IIC: This parameter specifies the starting column of the first line written to the OUTPUT file. This parameter is provided so that a section of commands (such as those appearing as input to COLLECT_TEXT) which are not normally to be formatted may be extracted from a PROCEDURE, formatted, and replaced in a formatted PROCEDURE.

Omission causes one to be used.

KEY_CHARACTER, KC: This parameter specifies the character which, if appearing in column one of the input line, causes the entire line to be written to the output file without any further processing.

Omission causes '*' to be used.

UTILITY_DEFINITION_FILE, UDF This parameter specifies the file which contains definitions for additional utility commands which may appear in the input file. The utility names and terminators defined within this file are added to those built into the formatter.

Omission causes only the built-in utilities to be recognized.

PROCESS_COLLECT_TEXT, PCT: This parameter specifies whether lines within COLLECT_TEXT, etc., data are to be formatted (TRUE) or not (FALSE).

Omission causes FALSE to be used.

TRANSLATE, T: This parameter specifies whether the input to the formatter is to be translated from “old” SCL to “new” SCL (TRUE) or not (FALSE).

Omission causes FALSE to be used.

STATUS: See the section on Error Handling.

15.1 Input to the Formatter

The input to the formatter is a file containing SCL commands. Usually the input consists of one or more procedures; however it is not required that the input be complete PROCEDUREs. Any collection of SCL commands may be formatted, but SCL structure blocks within the input file must be complete or the formatter reports errors.

15.1.1 Pragmats

Pragmats are special SCL comments in the input file which control the formatting process. A pragmat is an SCL comment with $ (dollar sign) as the first character of the comment. That is a pragmat begins with the characters "$ (double quote, dollar sign) and is terminated with " (double quote) or end-of-line. A pragmat may begin in any character position of a line but only blank characters may precede it. The “command definition” of a pragmat (where "$ may be thought of as the “command”) is:

```
"$ (
      command, c: name
      format, f: (by_name) boolean
      format_expressions, format_expression, fe: (by_name) boolean
      )
```

COMMAND, C This parameter specifies the name of a utility command or a utility terminator. The purpose of this parameter is to inform the formatter that the utility name or terminator is “hidden” (such as being included in a file). Upon encountering a pragmat with this parameter, the formatter proceeds as though the name had been encountered in the input and adjusts indentation accordingly.

Omission causes no change in indentation.

FORMAT, FMT, F: This parameter specifies whether formatting is enabled after the pragmat is processed. If there is a command on the line after the pragmat comment and separated from it by a semicolon, the specified action applies to that command in addition to following lines.

Omission causes no change with respect to formatting being enabled. This option is initialized to ON when the formatter starts.

FORMAT_EXPRESSIONS, FORMAT_EXPRESSION, FE: (*) This parameter specifies formatting action related solely to expressions after the pragmat is processed. It is provided for those situations where use is being made of application value types of which the formatter has no knowledge and considers to be in error. Setting this option to OFF allows the bulk of commands and statements in which the application values are used to still be formatted, i.e. you need not resort to turning off formatting altogether. The selected value of this option is ignored if formatting is disabled.

Omission causes no change with respect to formatting expressions. This option is initialized to ON when the formatter starts.

15.2 Utility Definition File

The formatter maintains a table of known utility names and corresponding terminators. The utilities initially entered into this table are:

Utility Names	Terminator Names
ADMINSTER_USER ADMU	QUIT
ANALYZE_DUMP ANAD	QUIT
BACKUP_PERMANENT_FILES BACKUP_PERMANENT_FILE BACPF	QUIT
BUILD_DIALOG_PROCESSOR BUIDP	QUIT QUI
BUILD_REAL_MEMORY	QUIT

BUIRM	
CREATE_APPLICATION_MENU CREAM	END_APPLICATION_MENU QUIT QUI
CREATE_INTERSTATE_CONNECTION CREIC	DELETE_INTERSTATE_CONNECTION DELIC QUIT QUI
CREATE_MESSAGE_MODULE CREMM	END_MESSAGE_MODULE ENDMM QUIT QUI
CREATE_OBJECT_LIBRARY CREOL	QUIT QUI
DEFINE_CLIENT DEFC	END_DEFINE_CLIENT ENDDC QUIT QUI
DEFINE_SERVER DEFS	END_DEFINE_SERVER ENDDS QUIT QUI
DISPLAY_BINARY_LOG DISBL	QUIT QUI
EDIT_DECK EDID	END QUIT QUI
EDIT_FILE EDIF	END QUIT QUI
EMAIL EMA	QUIT QUI
LINK_VIRTUAL_ENVIRONMENT LINVE	QUIT QUI
MAIL MAI	QUIT QUI
MANAGE_FILE_SERVER MANFS	QUIT QUI
MANAGE_NETWORK_APPLICATIONS MANAGE_NETWORK_APPLICATION MANNA	QUIT QUI
MEASURE_PROGRAM_EXECUTION MEAPE	QUIT QUI
NETWORK_OPERATOR_UTILITY NETOU	QUIT QUI
RESTORE_PERMANENT_FILES RESTORE_PERMANENT_FILE RESPF	QUIT QUI
SOURCE_CODE_UTILITY SCU	END QUIT QUI

The user may augment this table by generating a utility definition file and specifying that file on the formatter call. The file must consist of one line entries with the following “parameter list.”

```
(
      names, name, n: list of name = $required
```

```
      terminators, terminator, t: list of name = (quit, qui)
      )
```

NAMES, NAME, N: This parameter specifies a list of names for one utility.

TERMINATORS, TERMINATOR, T: This parameter specifies a list of names which terminate the utility.

Omission causes (QUIT, QUI) to be used.

Utility names must be unique and may not be the same as any of the “built-in” names. If the terminator names of a utility include more than QUIT and QUI, all terminators must be specified.

```
Example: n=(my_utility, mu) t=(end,quit,qui)
         names=(your_utility, yu)
         its_utility stop
```

15.3 Output of the Formatter

The formatter generates lines in the output file according to the following conventions.

1. If any input statement contains continuation lines, all continuation lines are read and concatenated with the beginning line of the statement before formatting of that statement begins.
2. The “current indent column” governs the starting column of the statement written to the output file. Initially the current indent column is column one (unless overridden by the initial_indent_column in the call to the formatter).
3. If the statement to be output (starting at the current indent column) is of such length as to exceed the specified page width, that statement is broken at, if possible, a “reasonable place” and as many continuation lines as required are generated. Breaking a line at a “reasonable place” means that items which are closely related to adjoining items are not placed on separate lines unless there is no alternative.
4. A continuation line is be indented six columns from the indent column of the first line of the statement.
5. If an expression consisting of <operand><operator><operand> must be broken, it is broken after the operator.
6. In certain cases (a string too long to fit on the entire line, for example) indentation must be suppressed on the continuation line.
7. If a statement consists of an SCL control statement which defines the beginning of an SCL structure block (IF, for example), all statements following such a statement are indented two spaces from the control statement until the corresponding end statement (IFEND, for example) is encountered.
8. If a statement is labeled, the label (followed by the continuation ellipses) is placed on a line by itself indented two columns less than the current indent column unless, of course, the current indent column is less than or equal to two.
9. SCL control statement identifiers (IF, WHILE, etc.) along with certain control words (DO, EXIT, etc.) are written to the output file in uppercase. All other names, except known utility names and terminators, are written in lowercase.
10. Utilities are treated as SCL structure blocks.
11. SCL comments are retained and written to the output file. Comments at the beginning of a statement (before the command name) are written as separate lines and indented to the current indent column. An exception to this is comments starting in column one of the input file. Such comments are written starting in column one of the output file.
12. Spaces are added or deleted as follows:
    a. A comma is written with no leading space and with a trailing space.
    b. Contiguous spaces on input are written as one space.
    c. Spaces are written both before and after the equals sign of an assignment statement.
    d. No spaces are written either before or after the equals sign that separates the name of a parameter from its value.
    e. No rule in this sub-section applies to characters within a string expression or comment.
13. If multiple statements, separated by semi-colons, appear on an input line, each statement is formatted on separate lines and the semi-colons discarded.
14. If a statement terminator is optional (THEN of an IF statement, for example), the formatter generates the terminator if it

does not appear on the input statement.

15. COLLECT_TEXT (COLT) is treated as a special case since the text being collected may not be SCL statements. Whenever a collect_text command is encountered, the parameter list is scanned for the value of the UNTIL (U) parameter. If the UNTIL parameter is not specified, or is specified as a literal string, formatting is turned off until the line containing the specified string (or **) is read. All collect_text lines are copied to the output file without any other processing. If the UNTIL parameter is specified other than as a literal string, or if the PROCESS_COLLECT_TEXT formatter parameter was specified as TRUE, formatting is not turned off (since there is no way of knowing when it is to be re-activated) and the results may be undefined. The collect_text command is written starting in column one with the command name written in uppercase.

    The MANAGE_REMOTE_FILES command is treated the same way as the COLLECT_TEXT command:

    The following subcommands of the CREATE_MESSAGE_MODULE subutility of the CREATE_OBJECT_LIBRARY utility are treated the same way as the COLLECT_TEXT command:

    CREATE_BRIEF_HELP_MESSAGE
    CREATE_FULL_HELP_MESSAGE
    CREATE_PARAMETER_ASSIST_MESSAGE
    CREATE_PARAMETER_HELP_MESSAGE
    CREATE_PARAMETER_PROMPT_MESSAGE
    CREATE_STATUS_MESSAGE

16. Any input line beginning with the key_character specified in the call to the formmatter is written directly to the output file without any other processing attempted.
17. If a statement formatted for output exceeds the maximum command size (65535 characters), any indentation present in the output line(s) of the statement is decremented until the maximum command size is not exceeded. If deleting all indentation for the statement does not result in an acceptable command size, a comment is issued to the output file and the input line(s) of the statement written instead of the formatted line(s).
18. If a blank line does not follow the last line of a procedure delcaration, a blank line is added. If a blank line does not appear before a PROCEND or FUNCEND statement, a blank line is added.
19. Certain errors are reported to the output file in the standard status message format.
20. If the formatter encounters a problem with a statement (such as not being able to determine the UNTIL value of COLLECT_TEXT), a a warning message is written to the output file as a comment.
21. For each error or warning message written to the output file, the input line(s) in which the error was detected immediately follow the message.
22. Certain information about a line in error may be retained by the formatter. For example, if a “WHEN” statement is recognized but an error is detected in some remaining portion of the statement, a WHEN block is still established and requires a WHENEND statement.
23. Whenever a statement specifying an end of a block (such as WHILEND) is encountered, the formatter checks that the block exists. If it does and if any block was started within the block being terminated, a formatter error message is generated indicating the line number of the statement beginning the block. This means, for example, that if a PROCEND statement is encountered (and a PROCEDURE block exists), then any unterminated blocks are noted in an error message.
24. In checking syntax and the like, the formatter assumes no knowledge of data types for parameters and expressions. for example, the statement

```
IF 'this is a string' = 255(10) THEN
```

    is considered to be acceptable. That is, the formatter detects only syntax errors and not errors of usage.

25. A count of error and warning messages issued is maintained and reported, if non-zero, as the formatter's termination status. If the error count exceeds a certain threshold (100), the formatter gives up, assuming that it is probably processing unknown text.

15.4 An Example

Input

```
procedure   aaa (
input,i : file = $REQUIRED
output, o : file=$OUTPUT
status)
put_this _message_out 'this is a message to be put to ..
output' o=$output
edit_file xyz
l,,a
include_file abcc
 "$ quit
while_label: while abc>def do
if hij='???' then
create_object_library
add_module library=:nve.abcdefghijklmnopqrstuvwxyz1.abcdefghi..
jklmn2
generate_library a
quit
else;exit
ifend;whilend;procend
```

Output

```
PROCEDURE aaa (
  input, i: file = $required
  output, o: file = $output
  status)

  put_this _message_out 'this is a message to be put to output' ..
        o=$output
  EDIT_FILE xyz
    l,, a
    include_file abcc
  "$ quit
while_label: ..
  WHILE abc > def DO
    IF hij = '???' THEN
      CREATE_OBJECT_LIBRARY
        add_module library=:nve.abcdefghijklmnopqrstuvwxyz1..
.abcdefghijklmn2
        generate_library a
      QUIT
    ELSE
      EXIT
    IFEND
  WHILEND while_label

PROCEND aaa
```

16 Commands Related to Variables

16.1 create_default_variable (credv)

The purpose of this command is to create a default variable and give it an initial value.

A default variable is an environment variable which is intended to be referenced via the <default name> specification in command or function parameters.

The type of a default variable is string. The contents of the string is interpreted as an expression for the parameter that references the default variable when no value is supplied for the parameter.

Apart from its usage there is nothing special about a default variable as contrasted with other environment variables. This command is provided to make creation of default variables more convenient than it might otherwise be.

```
create_default_variable, credv (
      name, n: data_name = $required
      default, d: string = $required
      scope, s: (by_name) key
          (environment, e)
          (job, j)
          (task, t)
          (utility, u)
        keyend = job
      status)
```

NAME, N: This parameter specifies the name of the variable to be created.

DEFAULT, D: This parameter specifies the string to be associated with the default variable.

SCOPE, S: This parameter specifies the scope of the default variable. See the section on Scope of Environment Variables for information about these options.

Omission causes JOB to be assumed.

STATUS: See the section on Error Handling.

16.2 create_file_variable (crefv)

The purpose of this command is to create a file variable and give it an initial value. The file variable can subsequently be used as an alias for the file or catalog designated by its value.

This command is provided to make creation of file aliases more convenient than it might otherwise be.

```
create_file_variable, crefv (
      name, n: data_name = $required
      file, f: file = $required
      scope, s: (by_name) key
          (environment, e)
          (job, j)
          (task, t)
          (utility, u)
        keyend = job
      status)
```

NAME, N: This parameter specifies the name of the file variable to be created.

FILE, F: This parameter specifies the file or catalog to which the variable is set.

SCOPE, S: This parameter specifies the scope of the file variable. See the section on Scope of Environment Variables for information about these options.

Omission causes JOB to be assumed.

STATUS: See the section on Error Handling.

16.3 create_variable_list_entry(ies (crevle) (*)

This command adds entries to either the beginning or end of the variable library list.

```
create_variable_list_entry, create_variable_list_entries, crevle (
      entry, entries, e: list of any of
          key
            $system
          keyend
          file
        anyend = $required
      placement, p: (by_name) key
          (after, a)
          (before, b)
        keyend = before
      status)
```

ENTRY, ENTRIES, E: This parameter specifies entries to be added to the variable list. They will appear in the variable list in the order specified. The $SYSTEM keyword is used to reference the variable library that is provided by the system which holds, for example, the variables that represent the names of the run-time libraries for the standard compilers.

PLACEMENT, P: This parameter specifies whether the entries added to the variable list are placed BEFORE or AFTER the current entries.

Omission will cause BEFORE to be used.

STATUS: See the section on Error Handling.

16.4 delete_variable(s) (delv)

This command deletes procedure or environment variables.

If an XREF'ed procedure variable is requested to be deleted, it is only the XREF'ed access that is removed, the original XDCL varaible is not affected. Yhis is also true in those cases were a scope of JOB, TASK or UTILITY was used to create a variable and the designated block was not the referencing block; i.e. a delete_variable request will only delete the access to the variable from the referencing block.

A procedure variable can only be deleted from within the block in which it was created or from within any block that is statically linked to the block in which it was created. See the section on Accessibility of Variables for more information.

```
delete_variable, delete_variables, delv (
      name, names, n: any of
          data_name
          list of data_name
        anyend = $required
      status)
```

NAME, NAMES, N: This parameter specifies the names of the variable(s) to be deleted from the current block.

STATUS: See the section on Error Handling.

16.5 delete_variable_list_entry(ies) (delvle) (*)

This command deletes entries from the variable list.

```
delete_variable_list_entry, delete_variable_list_entries, delvle (
      entry, entries, e: list of any of
          key
            all
            $system
          keyend
          file
        anyend = $required
      status)
```

ENTRY, ENTRIES, E: This parameter specifies the entries to be deleted from the variable list. If this parameter is specified as ALL, then all entries in the variable list are deleted. If the $SYSTEM keyword is used, then the variable library that is provided by the system is deleted from the variable list.

STATUS: See the section on Error Handling.

16.6 display_variable_list (disvl) (*)

NOTE there is a command in the system now called display_variable_list. That command will eventually become display_variable_list_entry.

This command displays the names of the entries in the variable list.

```
display_variable_list, disvl (
      output, o: file = $output
      status)
```

OUTPUT, O: This parameter designates the file to which the information is written.

Omission causes $output to be used.

STATUS: See the section on Error Handling.

16.7 display_variable_list_entry(ies) (disvle) (*)

This command displays information about one or more variable list entries.

```
display_variable_list_entry, display_variable_list_entries, disvle (
      entry, entries, e: list of any of
          key
            all
            (all_libraries, al)
            job
            $system
          keyend
          file
        anyend = job
      display_options, display_options, do: key
          (names, name, n)
          (values, value, v)
          (specifications, specification, s)
          (descriptions, description, d)
        keyend = (names, descriptions)
      output, o: file = $output
      status)
```

ENTRY, ENTRIES, E: This parameter selects the variable list entry or entries to be displayed. Any entry that can be put in the variable list can be selected regardless of whether it is presently in the variable list. The keyword ALL selects all entries in the variable list. The keyword ALL_LIBRARIES (AL) selects all libraries in the variable list. The keyword JOB selects

all of the variables available within the job that are not in libraries. The keyword $SYSTEM refers to the variable library provided by the system.

Omission of this parameter causes job to be selected.

DISPLAY_OPTIONS, DISPLAY_OPTION, DO: This parameter selects the form and content of the output.

NAMES, NAME, N: This option causes the names of all variables in the selected entries to be displayed.

VALUES, VALUE, V: This option specifies that the names and values of each variable in the selected entries are to be displayed.

SPECIFICATIONS, SPECIFICATION, S: This option specifies that all information about each variable in the selected entries is to be displayed. The information is produced in the form of a complete <variable definitions> specification which includes the types, values and any other attributes of the variables.

DESCRIPTIONS, DESCRIPTION, D: This option specifies that the descriptions associated with the variables be displayed. This option is ignored for variables in the JOB entry.

Omission of this parameter causes NAMES and DESCRIPTIONS to be selected.

OUTPUT, O: This parameter designates the file to which the information is written.

Omission causes $output to be used.

status: See the section on Error Handling.

17 Create_Variable_Library(ies) (crevl) (*)

This command utility is used to create and/or change libraries of SCL variables.

```
create_variable_library, create_variable_libraries, crevl (
      status)
```

STATUS: See the section on Error Handling.

A variable library is built up from other such libraries and/or selecting variables from the current command environment. The utility operates on a “working library” which is initially empty. Variables may be added to the working library, replaced on it or deleted from it via utility subcommands described below. Once the working library has the desired contents, an actual library may be generated from it via the generate_library subcommand. Upon completion, the generate_library command resets the working library to its initial empty state ready for the construction of another library, or the utility may be exited.

Any variable that can be referenced from within the utility may be incorporated into a library. The scope attributes of the selected variables are ignored; all variables on libraries are considered to be environment variables. Any other attributes of the variables are retained, e.g. READ, DEFER.

17.1 add_variable(s) (addv) (*)

This command adds selected variables from the current command environment to the working library. None of the selected variables may already be present in the working library.

```
add_variable, add_variables, addv (
      variable, variables, v: any of
          data_name
          list of data_name
        anyend = $required
      description, descriptions, d: list of string
      status)
```

VARIABLE, VARIABLES, V: This parameter specifies the variables to be added to the working library.

DESCRIPTION, DESCRIPTIONS, D This parameter specifies descriptions for the variables being added. The strings in this list are associated with the corresponding variables.

Omission causes no descriptions to be associated with the variables.

status: See the section on Error Handling.

17.2 combine_variable(s) (comv) (*)

This command adds selected variables from the current command environment to the working library. If a selected variable is already present in the working library, it replaces the variable on the library, otherwise it is added to the library.

```
combine_variable, combine_variables, comv (
      variable, variables, v: any of
          data_name
          list of data_name
        anyend = $required
      description, descriptions, d: list of string
      status)
```

VARIABLE, VARIABLES, V This parameter specifies the variables to be added to or replaced on the working library.

DESCRIPTION, DESCRIPTIONS, D: This parameter specifies descriptions for the variables being added or replaced. The

strings in this list are associated with the corresponding variables.

Omission causes no descriptions to be associated with variables being added, and leaves descriptions unchanged for variables being replaced.

STATUS: See the section on Error Handling.

17.3 replace_variable(s) (repv) (*)

This command replaces selected variables on the working library with variables from the current command environment. All of the selected variables must already be present in the working library.

```
replace_variable, replace_variables, repv (
      variable, variables, v: any of
          data_name
          list of data_name
        anyend = $required
      description, descriptions, d: list of string
      status)
```

VARIABLE, VARIABLES, V: This parameter specifies the variables to be replaced on the working library.

DESCRIPTION, DESCRIPTIONS, D: This parameter specifies descriptions for the variables being replaced. The strings in this list are associated with the corresponding variables.

Omission causes descriptions to be left unchanged.

STATUS: See the section on Error Handling.

17.4 add_library (addl) (*)

This command adds variables from a variable library to the working library. All or some of the variables on the library may be selected. None of the selected variables may already be present in the working library.

```
add_library, addl (
      library, l: file = $required
      variable, variables, v: any of
          key
            all
          keyend
          data_name
          list of data_name
        anyend = all
      status)
```

LIBRARY, L: This parameter specifies the library from which variables are to be added to the working library.

VARIABLE, VARIABLES, V: This parameter specifies the variables to be added to the working library. The keyword ALL may be used to specify that all variables on the designated library are to be added.

Omission causes ALL to be used.

STATUS: See the section on Error Handling.

17.5 combine_library (coml) (*)

This command combines variables from a variable library with the working library. All or some of the variables on the library may be selected. Any of the selected variables that are already present in the working library replace their replace the working library copy.

```
combine_library, coml (
      library, l: file = $required
      variable, variables, v: any of
          key
            all
          keyend
          data_name
          list of data_name
        anyend = all
      status)
```

LIBRARY, L: This parameter specifies the library from which variables are to be combined with the working library.

VARIABLE, VARIABLES, V: This parameter specifies the variables to be combined with the working library. The keyword ALL may be used to specify that all variables on the designated library are to be added.

Omission causes ALL to be used.

STATUS: See the section on Error Handling.

17.6 replace_library (repl) (*)

This command replaces variables on the working library with variables from a variable library. All or some of the variables on the library may be selected. All of the selected variables must already be present in the working library.

```
replace_library, repl (
      library, l: file = $required
      variable, variables, v: any of
          key
            all
          keyend
          data_name
          list of data_name
        anyend = all
      status)
```

LIBRARY, L: This parameter specifies the library from which variables are selected to be replaced on the working library.

VARIABLE, VARIABLES, V: This parameter specifies the variables to be replaced to the working library. The keyword ALL may be used to specify that all variables on the designated library are to replace variables on the working library.

Omission causes ALL to be used.

STATUS: See the section on Error Handling.

17.7 change_variable_description (chavd) (*)

This command changes the description associated with a variable on the working library.

```
change_variable_description, chavd (
      variable, v: data_name = $required
      description: string = $required
      status)
```

VARIABLE, V This parameter specifies the variables to be deleted from the working library.

DESCRIPTION, D: This parameter specifies a description the specified variable.

STATUS: See the section on Error Handling.

17.8 delete_variable(s) (delv) (*)

This command deletes selected variables from the working library.

```
delete_variable, delete_variables, delv (
      variable, variables, v: any of
          data_name
          list of data_name
        anyend = $required
      status)
```

VARIABLE, VARIABLES, V: This parameter specifies the variables to be deleted from the working library.

STATUS: See the section on Error Handling.

17.9 display_library (disl) (*)

This command displays information about the variables on the working library.

```
display_library, disl (
      display_options, display_options, do: key
          (names, name, n)
          (values, value, v)
          (specifications, specification, s)
          (descriptions, description, d)
        keyend = (names, descriptions)
      output, o: file = $output
      status)
```

DISPLAY_OPTIONS, DISPLAY_OPTION, DO: This parameter selects the form and content of the output.

NAMES, NAME, N: This option causes the names of all variables in the working library to be displayed.

VALUES, VALUE, V: This option specifies that the names and values of each variable in the working library are to be displayed.

SPECIFICATIONS, SPECIFICATION, S: This option specifies that all information about each variable in the working library is to be displayed. The information is produced in the form of a complete <variable definitions> specification which includes the types, values and any other attributes of the variables.

DESCRIPTIONS, DESCRIPTION, D: This option specifies that the descriptions associated with the variables be displayed.

Omission of this parameter causes NAMES and DESCRIPTIONS to be selected.

OUTPUT, O: This parameter designates the file to which the information is written.

Omission causes $output to be used.

STATUS: See the section on Error Handling.

17.10 generate_library (genl) (*)

This command generates a variable library from the working library. Upon successful completion the working library is reset to its empty state.

```
generate_library (genl)
      library, l: file = $required
      status)
```

LIBRARY, L: This parameter specifies the file to which the variable library is to be written.
STATUS: See the section on Error Handling.

17.11 quit (qui) (*)

This command exits the create_variable_library utility. It has no parameters.

```
quit, qui (
      )
```

18 Miscellaneous Control Commands

18.1 change_interaction_style (chais)

This command determines how applications, including SCL parameter dialogs, that are capable of both LINE and SCREEN mode interaction are to interact (by default). See the section on interactive usage for information on the LINE and SCREEN interaction styles used by the SCL interpreter.

```
change_interaction_style, chais (
      style, s: key
          (line, l)
          (screen, s)
        keyend = $required
      status)
```

STYLE, S: This parameter specifies the interaction style being selected.

STATUS: See the section on Error Handling.

18.2 change_message_level (chaml)

This command determines the amount of information to be included in subsequent status messages. See the section on message levels for details on the meaning of the various levels.

```
change_message_level, chaml (
      level, l: key
          (brief, b)
          (full, f)
        keyend = $required
      status)
```

LEVEL, L: This parameter specifies the message level being selected.

STATUS: See the section on Error Handling.

18.3 change_natural_language (chanl)

This command determines the preferred natural language to be used for messages, help information, etc.

```
change_natural_language, chanl (
      natural_language, nl: any of
          key
            danish
            dutch
            english
            finnish
            flemish
            french
            german
            italian
            norwegian
            portugese
            spanish
            swedish
            us_english
```

```
        keyend
        name
      anyend = $required
    status)
```

NATURAL_LANGUAGE, NL: This parameter specifies the natural language being selected.

STATUS: See the section on Error Handling.

18.4 change_scl_option(s) (chasclo, chaso)

This command determines the values of certain options for the System Command Language (SCL) interpreter.

```
change_scl_option, change_scl_options, chasclo, chaso (
      line_style_correction_prompts, lscp: (BY_NAME) key
          (line, l)
          (screen, s)
          none
        keyend = $optional
      screen_style_correction_prompts, sscp: (BY_NAME) key
          (screen, s)
          none
        keyend = $optional
      name_folding_level, nfl: (BY_NAME) key
          (standard_folding, sf)
          (full_folding, ff)
        keyend = $optional
      wild_card_pattern_type, wcpt: (BY_NAME) key
          (basic, b)
          (extended, e)
        keyend = $optional
      status)
```

LINE_STYLE_CORRECTION_PROMPTS, LSCP: This parameter specifies how the SCL interpter will prompt the user when a command entered interactively in LINE mode is given an incorrect parameter or a required parameter is omitted. Such prompting for parameter correction can be selected to occur in LINE mode, SCREEN mode or not at all (NONE). See “Interactive Usage” in section 1 of this ERS for more information.

Omission causes the LINE_STYLE_CORRECTION_PROMPTS option to be left unchanged. The initial value for this option is LINE.

SCREEN_STYLE_CORRECTION_PROMPTS, SSCP: This parameter specifies how the SCL interpter will prompt the user when a command entered interactively in SCREEN mode is given an incorrect parameter or a required parameter is omitted. Such prompting for parameter correction can be selected to occur in SCREEN mode or not at all (NONE). See “Interactive Usage” in section 1 of this ERS for more information.

Omission causes the SCREEN_STYLE_CORRECTION_PROMPTS option to be left unchanged. The initial value for this option is SCREEN.

NAME_FOLDING_LEVEL, NFL: This parameter specifies which characters allowed in NOS/VE names are considered to be letters with both lower and upper case versions. When a name is specified, any lower case letters in it are “folded” to their upper case counterparts.

STANDARD_FOLDING, SF: This option designates that only the twenty-six letters a..z are to be folded to A..Z.

FULL_FOLDING, FF: This option designates that in addition to the standard folding, the following folding is to occur:

Lower Case Character	Upper Case Character
@	`
[	{
\	|

]	}
^	~

This option affects the interpretation of *all* "SCL" names including, files names, variable and parameter names, names used within the Source Code Utility and the Object Code Utilities, etc.

Omission causes the NAME_FOLDING_LEVEL option to be left unchanged. The initial value for this option is STANDARD_FOLDING.

WILD_CARD_PATTERN_TYPE, WCPT: This parameter specifies whether "wild card" patterns are to be treated as BASIC or EXTENDED patterns when used in file references. A number of functions such as $SELECT_WILD_CARD_NAMES also use this option as the default value for their corresponding parameters.

See the section on "Wild Card" Patterns for information on the two types of "wild card" patterns.

Omission causes the WILD_CARD_PATTERN_TYPE option to be left unchanged. This initial value for this option is EXTENDED.

STATUS: See the section on Error Handling.

18.5 change_unseen_mail_action (chauma)

This command establishes the default action to be taken by the system when an UNSEEN_MAIL condition arises, i.e. when a "letter" is sent, via Mail/VE, to a mailbox owned by the user on whose behalf the job is running.

```
change_unseen_mail_action, chauma (
      action, a: key
          (display, d)
          (post, p)
        keyend = $required
      status)
```

ACTION, A: This parameter specifies the action to be taken.

DISPLAY, D: specifies that the system is to display a message to file $RESPONSE indicating the arrival of mail

POST, P specifies that the system is to "post" or "hold onto" the unseen mail condition until the UNSEEN_MAIL_ACTION is changed to or reverts to "display."

STATUS: See the section on Error Handling.

18.6 change_working_catalog (chawc)

This command establishes the working catalog to be used in the evaluation of subsequent file expressions given in the form of relative paths. See the section on file expressions for information about relative paths, absolute paths and file expressions.

```
change_working_catalog, chawc (
      catalog, c: file = $required
      status)
```

CATALOG, C: This parameter specifies the new working catalog.

STATUS: See the section on Error Handling.

18.7 collect_text (colt)

The purpose of this command is to write to a specified file, the lines of text that follow the command. The text is collected (read) from $COMMAND until a line containing only a termination string is encountered. If no such line is found, the end-of-

information (or an end-of-partition) terminates text collection with a warning status.

If COLLECT_TEXT creates the output file it sets its FILE_CONTENTS attribute to LEGIBLE_DATA, and its PAGE_FORMAT attribute to UNTITLED. Other attributes will have the standard defaults.

If the output file already exists, its FILE_CONTENTS attribute must be one of LIST, LEGIBLE_DATA, LEGIBLE-SCL_INCLUDE, LEGIBLE_SCL_JOB, LEGIBLE_SCL_PROCEDURE or UNKNOWN.

```
collect_text, colt (
      output, o: file = $required
      until, u: string = '**'
      prompt, p: (by_name) string 0..31 = ' ct? '
      substitution_mark, sm: (by_name) any of
          key
            none
          keyend
          string 1
        anyend = none
      input, i: (by_name) file = $optional
      status)
```

OUTPUT, O: This parameter specifies the file to which the collected text is to be written.

UNTIL, U: This parameter specifies the string which, if it appears alone on a line, will terminate the collection of text.

Omission causes '**' to be used.

PROMPT, P: This parameter specifies the prompt string to be issued for each line if collect_text is getting its input from an interactive terminal. If the null string is specified, no prompt is issued. The first character in the prompt is a format effector character. Typically a space character is used which causes each prompt to be issued on a new line.

Omission causes ' ct? ' to be used.

SUBSTITUTION_MARK, SM: This parameter specifies a character that is used within the text to be collected to delimit text to be substituted, or that no such character be used (NONE). Corresponding pairs of substitution marks must appear on the same line. The text between such marks is evaluated as an SCL string expression, the result of which replaces the original text including the substitution marks.

If the expression cannot be evaluated or its result cannot be converted to a string, the COLLECT_TEXT command terminates with an appropriate error. If no second substitution mark is found on a line, the end-of-line is treated as the second substitution mark. If two consecutive substitution marks appear, they are replaced by a single substitution mark in the collected text. Substitution cannot be used to construct the line containing the termination string, i.e. the termination string is checked for prior to processing the line for substitution.

Omission causes NONE to be used.

INPUT, I: This parameter can be used to specify the file from which the text is to be collected.

The FILE_CONTENTS attribute of this file must be either LEGIBLE_DATA, LEGIBLE_SCL_INCLUDE, LEGIBLE_SCL_JOB, LEGIBLE_SCL_PROCEDURE or UNKNOWN.

Omission causes the text to be collected from the current source of command input, i.e. $COMMAND. This parameter should only be used to specify a file other than $COMMAND.

STATUS: See the section on Error Handling.

18.8 display_command_information (disci)

This command displays information about a command and its parameters. The information available includes the names for the command and the names of the parameters, their types and their default values. If a “help module” exists for the command, the requested information from it is displayed, otherwise the information is derived directly from the command's parameter description table.

This command is not intended to be a replacement for information in on-line or hard copy manuals, but rather a “memory jogger” for any command. It will work for any command that could be called at the point where this command is called (including system supplied commands, SCL command procedures, user programs and utility subcommands).

This command can be used to obtain information about the parameters for any command that uses SCL services to parse its parameters.

```
display_command_information, disci (
      command, c: list of command_reference = $required
      output, o: file = $output
      display_options, display_option, do: list of key
          all
          (all_names, all_name, an)
          (source, s)
          (brief_help, bh)
          (full_help, fh)
          (compact_parameter_descriptions, compact_parameter_description, cpd)
          (parameter_descriptions, parameter_description, pd)
          (parameter_help, ph)
          (advanced_usage, au)
        keyend = osd$disci_display_options,
              (all_names, brief_help, compact_parameter_descriptions)
      status)
```

COMMAND, C: This parameter specifies the commands for which information is sought.

OUTPUT, O: This parameter specifies the file to which the parameter information is written.

Omission will cause $output to be used.

DISPLAY_OPTIONS, DISPLAY_OPTION, DO: This parameter specifies the type of information to be displayed.

ALL_NAMES, ALL_NAME, AN: This option selects that all of the names for the command, i.e. its full (nominal) name, its aliases and its abbreviation, be displayed.

SOURCE, S: This option selects that the source of the command be displayed. A command's source indicates the “command list entry” in which the command was found.

BRIEF_HELP, BH: This option selects that “brief help” message (if any) for the command from its help module (if any) be displayed.

FULL_HELP, FH: This option selects that “full help” message (if any) for the command from its help module (if any) be displayed. If there is no “full help” message but there is a “brief help” message, it is shown. If the brief_help option is selected as well as the full_help option, the brief_help option is ignored.

COMPACT_PARAMETER_DESCRIPTIONS, COMPACT_PARAMETER_DESCRIPTION, CPD: This option selects that the information about the command's parameters obtained from the command's parameter description table be displayed in a compact format.

PARAMETER_DESCRIPTIONS, PARAMETER_DESCRIPTION, PD: This option selects that the information about the command's parameters obtained from the command's parameter description table be displayed.

PARAMETER_HELP, PH: This option selects that the “parameter help” messages for the command's parameters be displayed. If there is no “parameter help” message for a particular parameter, the information about that parameter obtained from the command's parameter description table is displayed.

ADVANCED_USAGE, AU: This option selects that information about “advanced” parameters and keywords be displayed. If this option is selected but neither the parameter_descriptions nor the parameter_help option is selected, the parameter_descriptions option is automatically selected.

ALL: This option selects all of the above options.

Omission will cause ALL_NAMES, BRIEF_HELP and COMPACT_PARAMETER_DESCRIPTIONS to be used. This can be overridden by creating a “default variable” called OSD$DISCI_DISPLAY_OPTIONS and putting a string representing your favorite options into it.

STATUS: See the section on Error Handling.

18.9 display_function_information (disfi)

This command displays information about a function and its parameters. The information available includes the names for the function and the names of the parameters, their types and their default values. If a “help module” exists for the function, the requested information from it is displayed, otherwise the information is derived directly from the command's parameter description table.

This command is not intended to be a replacement for information in on-line or hard copy manuals, but rather a “memory jogger” for any function. It will work for any function that could be called at the point where this command is called (including system supplied functions, SCL function procedures and utility supplied functions).

This command can be used to obtain information about the parameters for any function that uses SCL services to parse its parameters.

```
display_function_information, disfi (
      function, f: list of data_name = $required
      output, o: file = $output
      display_options, display_option, do: list of key
          all
          (all_names, all_name, an)
          (source, s)
          (brief_help, bh)
          (full_help, fh)
          (compact_parameter_descriptions, compact_parameter_description, cpd)
          (parameter_descriptions, parameter_description, pd)
          (parameter_help, ph)
          (advanced_usage, au)
        keyend = osd$disfi_display_options,
               (all_names, brief_help, compact_parameter_descriptions)
      status)
```

FUNCTION, F: This parameter specifies the functions for which information is sought.

OUTPUT, O: This parameter specifies the file to which the parameter information is written.

Omission will cause $OUTPUT to be used.

DISPLAY_OPTIONS, DISPLAY_OPTION, DO: This parameter specifies the type of information to be displayed.

ALL_NAMES, ALL_NAME, AN: This option selects that all of the names for the function, i.e. its full (nominal) name, its aliases and its abbreviation, be displayed.

SOURCE, S: This option selects that the source of the function be displayed. A function's source indicates the “command list entry” in which the function was found.

BRIEF_HELP, BH: This option selects that the “brief help” message (if any) for the function from its help module (if any) be displayed.

FULL_HELP, FH: This option selects that the “full help” message (if any) for the function from its help module (if any) be displayed. If there is no “full help” message but there is a “brief help” message, it is shown. If the brief_help option is selected as well as the full_help option, the brief_help option is ignored.

COMPACT_PARAMETER_DESCRIPTIONS, COMPACT_PARAMETER_DESCRIPTION, CPD: This option selects that the information about the command's parameters obtained from the command's parameter description table be displayed in a compact format.

PARAMETER_DESCRIPTIONS, PARAMETER_DESCRIPTION, PD: This option selects that the information about the function's parameters obtained from the function's parameter description table be displayed.

PARAMETER_HELP, PH: This option selects that the “parameter help” messages for the function's parameters be displayed. If there is no “parameter help” message for a particular parameter, the information about that parameter obtained from the function's parameter description table is displayed.

ADVANCED_USAGE, AU: This option selects that information about “advanced” parameters and keywords be displayed. If this option is selected but neither the parameter_descriptions nor the parameter_help option is selected, the parameter_descriptions option is automatically selected.

ALL: This option selects all of the above options.

Omission will cause ALL_NAMES, BRIEF_HELP and COMPACT_PARAMETER_DESCRIPTIONS to be used. This can be overridden by creating a "default variable" called OSD$DISFI_DISPLAY_OPTIONS and putting a string representing your favorite options into it.

STATUS: See the section on Error Handling.

18.10 display_command_stack (discs) (*)

This command displays the current command environment (stack). This environment is maintained in a stack which contains an entry for each active "block" in the environment.

```
display_command_stack, discs (
      display_options, display_option, do: list of key
          all
          (all_blocks, all_block, ab)
          (blocks, block, b)
          (command_mode, cm)
          (environment_variables, environment_variable, ev)
          (message_level, ml)
          (natural_language, nl)
          (procedure_variables, procedure_variable, pv)
          (variables, variable, v)
          (variable_values, variable_value, vv)
        keyend = (blocks, command_mode, message_level,
              environment_variables, variable_values)
      output, o: file = $output
      status)
```

DISPLAY_OPTIONS, DISPLAY_OPTION, DO: This parameter specifies the amount of information to be displayed.

ALL: All available information is displayed.

ALL_BLOCKS, ALL_BLOCK, AB: This option selects a display of all "blocks," including those created internally. This option may be of use to developers of command utilities for debugging purposes.

BLOCKS, BLOCK, B: This option selects a display of those blocks created directly by the user.

COMMAND_MODE, CM: This option selects the command mode to be displayed.

ENVIRONMENT_VARIABLES, ENVIRONMENT_VARIABLE, EV: This option selects a display of the names of all environment variables and their attributes.

MESSAGE_LEVEL, ML: This option selects the message level to be displayed.

NATURAL_LANGUAGE, NL: This option selects the natural language preferred for the job to be displayed.

PROCEDURE_VARIABLES, PROCEDURE_VARIABLE, PV: This option selects a display of the names of all procedure variables and their attributes.

VARIABLES, VARIABLE, V: This option selects a display of the names of all environment and procedure variables and their attributes.

VARIABLE_VALUES, VARIABLE_VALUE, VV: This option selects a display of the values of the variables selected by the other options. If no other variable selection options are specified, environment variables are assumed.

Omission causes a display of those items considered "most interesting" to most users.

OUTPUT, O: This parameter specifies the file to which the information will be displayed.

Omission causes $OUTPUT to be used.

STATUS: See the section on Error Handling.

18.11 display_value(s) (disv)

The purpose of this command is to display a list of values.

```
display_value, display_values, disv (
      values, value, v: any = $required
      output, o: file = $output
      display_options, display_option, do: list of key
          (elements, element, e)
          (data_structure, ds)
          (source, s)
        keyend = elements
      status)
```

VALUE, VALUES, V: This parameter specifies the value to be displayed.

OUTPUT, O: This parameter specifies the file to which the information will be displayed.

Omission causes $OUTPUT to be used.

DISPLAY_OPTIONS, DISPLAY_OPTION, DO: This parameter specifies the form and content of the output.

ELEMENTS, ELEMENT, E: each element of the value is displayed as a separate line in the output. Status values are formatted as messages and may therefore occupy more than one line. The built-in record types date_time, scu_line_identifier and time_increment are considered to be elements for purposes of this option.

DATA_STRUCTURE, DS: each element of the value is converted to a separate line in the output. The generic data type of the value (and components of the value, if applicable) are included in the output as "comments" prefixing the corresponding value or value component. Items that are components of data structures are prefixed with identifying "labels," e.g. field names for record elements, subscripts for array elements, etc.

SOURCE, S: the value is output in the form of an SCL "expression" that if evaluated with an appropriate type specification would yield exactly the same value. All lines in the output, except the last, will end with an ellipsis "..".

Omission causes ELEMENTS to be used.

STATUS: See the section on Error Handling.

18.12 display_working_catalog (diswc)

This command displays the working catalog.

```
display_working_catalog, diswc (
      output, o: file = $output
      status)
```

OUTPUT, O: This parameter specifies the file to which the working catalog will be displayed.

Omission causes $OUTPUT to be used.

STATUS: See the section on Error Handling.

18.13 get_line(s) (getl)

The purpose of this command is to read one or more line(s) from the specified file.

```
get_line, get_lines, getl (
      variable, v: (var) any of
          string
          list 0..$max_list of string
          array of string
```

```
  anyend = $required
input, i: file = $required
prompt, p: string
line_count, lc: (var) integer
status)
```

VARIABLE, V: This parameter specifies the variable into which the line(s) are to be read.

INPUT, I: This parameter specifies the file containing the line(s) to be read.

PROMPT, P: This parameter specifies the string to be used to prompt the user to enter the value. If the file specified by the input parameter is not assigned to an interactive terminal no prompt is issued.

The FILE_CONTENTS attribute of this file must be either LEGIBLE_DATA, LEGIBLE_SCL_INCLUDE, LEGIBLE_SCL_JOB, LEGIBLE_SCL_PROCEDURE or UNKNOWN.

If this parameter is omitted, the prompt string will consist of the word “ENTER” followed by the name of the variable into which the line will be placed.

LINE_COUNT, LC: This parameter specifies a variable into which is placed a count of the number of lines read by the get_line command.

Omission causes no LINE_COUNT variable to be written.

STATUS: See the section on Error Handling.

18.14 include_command (incc)

The purpose of this command is to interpret a string as one or more commands or control statements. The specified commands logically replaces the INCLUDE_COMMAND command. This command provides a facility for constructing a command (e.g. via string concatenation, or read from a file via GET_LINE) and having it processed.

The text passed to this command must conform to the <included line> syntax described in the section “Input to the SCL Interpreter.”

This command differs from the INCLUDE_LINE command in that the latter builds a whole new “input control block” in which to process the commands and control statements, whereas INCLUDE_COMMAND processes them in the context of the current “input control block.” This means that if commands from the current input stream are being logged, the included commands will also be logged; and, if the current input stream is an interactive one, the included commands will be treated as though they were entered from an interactive terminal.

```
include_command, incc (
      command, c: string = $required
      enable_echoing, enable_echo, ee: boolean = yes
      status)
```

COMMAND, C: This parameter specifies the command to be interpreted.

ENABLE_ECHOING, ENABLE_ECHO, EE: This parameter determines whether the command may be echoed (YES) or not (NO).

Omission causes YES to be used.

STATUS: See the section on Error Handling.

18.15 include_file (incf)

The purpose of this command is to include text contained in a specified file. The included text logically replaces the INCLUDE_FILE command. INCLUDE_FILE allows SCL statements contained in a specified file to be processed in the same context as the INCLUDE_FILE command; i.e. parameter substitution, variables and condition selections are the same.

This command can be used on its own or in conjunction with a command utility to initiate processing of the commands for the utility.

The text in the file must conform to the <included file> syntax described in the section “Input to the SCL Interpreter.”

The attributes of the included file must conform to the following constraints.

1. The file_contents attribute must be LEGIBLE_SCL_INCLUDE, LEGIBLE_DATA or UNKNOWN.
2. The file may not have the attribute combination record_type UNDEFINED (U) and block_type SYSTEM_SPECIFIED (SS).
3. The access_mode attribute must contain at least the read and/or execute modes.
4. The request must be issued from a ring that is less than or equal to the R2 component of the ring_attributes of the file. If the access_mode attribute does not include read (i.e. the file is “execute-only,” the request is also constrained to be issued from a ring greater than or equal to the R1 component of the ring attributes of the file.
5. A user defined file access procedure (fap) may be associated with the file provided that read is in the access_mode attribute.

```
include_file, incf (
      file, f: file = $required
      prompt, p: string
      utility, u: any of
          key
            none
          keyend
          name
        anyend = none
      status)
```

FILE, F: This parameter specifies the file to be included.

PROMPT, P: This parameter specifies a prompt string to be used if the file is assigned to an interactive terminal.

If used with a command utility, the utility's prompt is used.

Omission, when not used with a command utility, causes 'incf' to be used.

UTILITY, U: This parameter specifies the name of the utility with which the include_file is to be associated. NONE may be specified to avoid association with any utility.

Omission causes NONE to be assumed.

STATUS: See the section on Error Handling.

18.16 include_line (incl)

The purpose of this command is to interpret a string as a statement list. The specified statement list logically replaces the INCLUDE_LINE command. This command provides a facility for constructing a “line” of one or more statements (e.g. via string concatenation, or read from a file via GET_LINE) and having it processed.

This command can be used on its own or in conjunction with a command utility to initiate processing of the commands for the utility.

The text on the line must conform to the <included line> syntax described in the section “Input to the SCL Interpreter.”

This command differs from the INCLUDE_COMMAND command in that INCLUE_LINE builds a whole new “input control block” in which to process the commands and control statements, whereas INCLUDE_COMMAND processes them in the context of the current “input control block.” Since a new input block is created for this command, any structured statements being included must be complete, i.e. must both begin and end in the included line.

```
include_line, incl (
      statement_list, sl: string = $required
      enable_echoing, enable_echo, ee: boolean = yes
      utility, u: any of
          key
            none
          keyend
```

```
          name
        anyend = none
      status)
```

STATEMENT_LIST, SL: This parameter specifies the string to be interpreted.

ENABLE_ECHOING, ENABLE_ECHO, EE: This parameter determines whether the statement list may be echoed (YES) or not (NO).

Omission causes YES to be used.

UTILITY, U: This parameter specifies the name of the utility with which the include_line is to be associated. NONE may be specified to avoid association with any utility.

Omission causes NONE to be assumed.

STATUS: See the section on Error Handling.

18.17 put_line(s) (putl)

The purpose of this command is to write one or more lines to a file. If the file to which the lines are being written is a list file (i.e. its file_contents attribute is LIST) the first character of each string written is assumed to be the format effector for the corresponding line.

```
put_line, put_lines, putl (
      line, lines, l: list of string = $required
      output, o: file = $output
      status)
```

LINE, LINES, L: This parameter specifies the list of strings to be written.

OUTPUT, O: This parameter specifies the file to which the information will be written.

Omission causes $OUTPUT to be used.

STATUS: See the section on Error Handling.

18.18 wait

The wait command suspends command processing until one or more of a list of events has occurred. If more than one event is specified, the command will optionally terminate when any one or all of the events occur. This command only affects the task in which it was issued.

```
wait (
      time, t: any of
          integer 0..0ffffffff(16)
          time_increment
        anyend
      task_names, task_name, tn: (task_list) list of name
      queue_names, queue_name, qn: list of name
      until, u: key
          all
          any
        keyend = all
      status)
```

TIME, T: This parameter specifies the amount of time to wait before making the command sequence issuing the WAIT command eligible to resume processing. If an integer is specified it represents the number of milliseconds to wait. The actual time that elapses between issuing the command and resuming processing is dependent upon system load and job priority.

Omission will cause time to be ignored by this command.

TASK_NAMES, TASK_NAME, TN This parameter specifies a list of task names for which completion is to be awaited.

Omission will cause task completions to be ignored by this command.

QUEUE_NAMES, QUEUE_NAME, QN: This parameter specifies a list of queue names from which a message is to be awaited.

Omission will cause queues to be ignored by this command.

UNTIL, U: This parameter specifies whether the occurrence of ANY one of the specified events will terminate the command, or ALL of the specified events must occur before the command will terminate.

Omission will cause ALL to be used.

STATUS: See the section on Error Handling.