**Omega Draft Standard** Version 2

**1. Identification of the Proposed Change**

**1.1 Title: Omega Draft Standard** Version 2

**1.2 MDC Proposer and Sponsor:**

Editor: Frederick D. S. Marshall

1660 South Columbian Way (ISC)

Seattle, WA 98108-1597

O: (206) 764-2283 H: (425) 430-0932

fax: (206) 764-2923

toad@isc-sf.va.gov FORUM

**1.3 Motion:**

The proposer recommends that this document be accepted by SC16 as a Type C document.

**1.4 History:**

March 1998 X11/SC16/TG1/98-2 <Current Document>

First modern version of Omega Draft Standard.

Proposed as a SC16 Type C document.

**1.5 Dependencies**

Dependent on no existing X11.1 proposals.

**2. Justification of Proposed Change**

**2.1 Needs**

Members of the MDC have been trying to introduce OO into M since 1982.

**2.2 Existing Practice in Area of the Proposed Change**

To be added later: 1) object usage approaches, 2) OO languages that compile into M, 3) OO systems written in M.

**3. Description of Proposed Change**

**3.1 General Description of the Proposed Change**

See the Omega Design Goals document, to be integrated into this one later.

**3.2 Annotated Examples of Use**

To be added later. Very important.

**3.3 Formalization**

See below.

Table of Contents

1. Scope 1

2. Normative References 1

3. Conformance 2

3.1 Implementations 2

3.2 Programs 2

4. Definitions 3

5. Metalanguage Description 6

6. Routine routine 7

6.1 Routine head routinehead 7

6.2 Routine body routinebody 7

6.2.1 Level line levelline 8

6.2.2 Formal line formalline 8

6.2.3 Label label 8

6.2.4 Label separator ls 8

6.2.5 Line body linebody 8

6.3 Routine execution 9

6.3.1 Transaction processing 9

6.3.2 Error processing 10

6.4 Embedded programs 11

7. Expression expr 12

7.1 Expression atom expratom 12

7.1.1 Variables 12

7.1.2 Variable name glvn 13

7.1.2.1 Local variable name lvn 13

7.1.2.2 Local variable handling 13

7.1.2.3 Process-Stack 15

7.1.2.4 Global variable name gvn 16

7.1.3 Structured system variable ssvn 17

7.1.3.1 ^$CHARACTER 18

7.1.3.2 ^$DEVICE 19

7.1.3.3 ^$GLOBAL 20

7.1.3.4 ^$JOB 20

7.1.3.5 ^$LIBRARY 21

7.1.3.6 ^$LOCK 21

7.1.3.7 ^$ROUTINE 21

7.1.3.8 ^$SYSTEM 22

7.1.3.9 ^$Z[unspecified] 22

7.1.3.10 ssvns specifying default environments 22

7.1.4 Expression item expritem 23

7.1.4.1 String literal strlit 23

7.1.4.2 Numeric literal numlit 23

7.1.4.3 Numeric data values 23

7.1.4.4 Meaning of numlit 24

7.1.4.5 Numeric interpretation of data 24

7.1.4.6 Integer interpretation 25

7.1.4.7 Truth-value interpretation 25

7.1.4.8 Extrinsic function exfunc 25

7.1.4.9 Extrinsic variable exvar 26

7.1.4.10 Intrinsic special variable names svn 26

7.1.4.10.1 $DEVICE 27

7.1.4.10.2 $ECODE 27

7.1.4.10.3 $ESTACK 28

7.1.4.10.4 $ETRAP 28

7.1.4.10.5 $HOROLOG 28

7.1.4.10.6 $IO 28

7.1.4.10.7 $IOREFERENCE 28

7.1.4.10.8 $JOB 29

7.1.4.10.9 $KEY 29

7.1.4.10.10 $PIOREFERENCE 29

7.1.4.10.11 $PRINCIPAL 29

7.1.4.10.12 $QUIT 30

7.1.4.10.13 $REFERENCE 30

7.1.4.10.14 $STACK 30

7.1.4.10.15 $STORAGE 30

7.1.4.10.16 $SYSTEM 30

7.1.4.10.17 $TLEVEL 31

7.1.4.10.18 $TRESTART 31

7.1.4.10.19 $X 31

7.1.4.10.20 $Y 31

7.1.4.10.21 $Z 31

7.1.4.11 Unary operator unaryop 32

7.1.4.12 Name value namevalue 32

7.1.5 Intrinsic function function 32

7.1.5.1 $ASCII 33

7.1.5.2 $CHAR 33

7.1.5.3 $DATA 34

7.1.5.4 $EXTRACT 34

7.1.5.5 $FIND 35

7.1.5.6 $FNUMBER 35

7.1.5.7 $GET 36

7.1.5.8 $JUSTIFY 37

7.1.5.9 $LENGTH 37

7.1.5.10 $NAME 37

7.1.5.11 $ORDER 38

7.1.5.12 $PIECE 39

7.1.5.13 $QLENGTH 40

7.1.5.14 $QSUBSCRIPT 40

7.1.5.15 $QUERY 40

7.1.5.16 $RANDOM 41

7.1.5.17 $REVERSE 42

7.1.5.18 $SELECT 42

7.1.5.19 $STACK 42

7.1.5.20 $TEXT 43

7.1.5.21 $TRANSLATE 44

7.1.5.22 $VIEW 44

7.1.5.23 $Z 44

7.1.6 Omega Standard Library 44

7.1.6.1 Library definitions 44

7.1.6.2 Library Element Definitions 45

7.1.6.3 Availability of library elements 46

7.1.6.4 CHARACTER Library elements 46

7.1.6.4.1 $%COLLATE^CHARACTER 46

7.1.6.4.2 $%COMPARE^CHARACTER 46

7.1.6.5 MATH Library elements 47

7.1.6.5.1 $%ABS^MATH 47

7.1.6.5.2 $%ARCCOS^MATH 47

7.1.6.5.3 $%ARCCOSH^MATH 47

7.1.6.5.4 $%ARCCOT^MATH 47

7.1.6.5.5 $%ARCCOTH^MATH 47

7.1.6.5.6 $%ARCCSC^MATH 47

7.1.6.5.7 $%ARCSEC^MATH 48

7.1.6.5.8 $%ARCSIN^MATH 48

7.1.6.5.9 $%ARCSINH^MATH 48

7.1.6.5.10 $%ARCTAN^MATH 48

7.1.6.5.11 $%ARCTANH^MATH 48

7.1.6.5.12 $%CABS^MATH 48

7.1.6.5.13 $%CADD^MATH 48

7.1.6.5.14 $%CCOS^MATH 49

7.1.6.5.15 $%CDIV^MATH 49

7.1.6.5.16 $%CEXP^MATH 49

7.1.6.5.17 $%CLOG^MATH 49

7.1.6.5.18 $%CMUL^MATH 49

7.1.6.5.19 $%COMPLEX^MATH 49

7.1.6.5.20 $%CONJUG^MATH 49

7.1.6.5.21 $%COS^MATH 49

7.1.6.5.22 $%COSH^MATH 50

7.1.6.5.23 $%COT^MATH 50

7.1.6.5.24 $%COTH^MATH 50

7.1.6.5.25 $%CPOWER^MATH 50

7.1.6.5.26 $%CSC^MATH 50

7.1.6.5.27 $%CSCH^MATH 50

7.1.6.5.28 $%CSIN^MATH 50

7.1.6.5.29 $%CSUB^MATH 51

7.1.6.5.30 $%DECDMS^MATH 51

7.1.6.5.31 $%DEGRAD^MATH 51

7.1.6.5.32 $%DMSDEC^MATH 51

7.1.6.5.33 $%E^MATH 51

7.1.6.5.34 $%EXP^MATH 51

7.1.6.5.35 $%LOG^MATH 51

7.1.6.5.36 $%LOG10^MATH 52

7.1.6.5.37 $%MTXADD^MATH 52

7.1.6.5.38 $%MTXCOF^MATH 52

7.1.6.5.39 $%MTXCOPY^MATH 52

7.1.6.5.40 $%MTXDET^MATH 52

7.1.6.5.41 $%MTXEQU^MATH 52

7.1.6.5.42 $%MTXINV^MATH 52

7.1.6.5.43 $%MTXMUL^MATH 53

7.1.6.5.44 $%MTXSCA^MATH 53

7.1.6.5.45 $%MTXSUB^MATH 53

7.1.6.5.46 $%MTXTRP^MATH 53

7.1.6.5.47 $%MTXUNIT^MATH 53

7.1.6.5.48 $%PI^MATH 53

7.1.6.5.49 $%RADDEG^MATH 54

7.1.6.5.50 $%SEC^MATH 54

7.1.6.5.51 $%SECH^MATH 54

7.1.6.5.52 $%SIGN^MATH 54

7.1.6.5.53 $%SIN^MATH 54

7.1.6.5.54 $%SINH^MATH 54

7.1.6.5.55 $%SQRT^MATH 54

7.1.6.5.56 $%TAN^MATH 54

7.1.6.5.57 $%TANH^MATH 55

7.1.6.6 STRING Library Elements 55

7.1.6.6.1 $%PRODUCE^STRING 55

7.1.6.6.2 $%REPLACE^STRING 55

7.2 Expression tail exprtail 55

7.2.1 Binary operator binaryop 56

7.2.1.1 Concatenation operator 56

7.2.1.2 Arithmetic binary operators 56

7.2.2 Truth operator truthop 56

7.2.2.1 Relational operator relation 57

7.2.2.2 Numeric relations 57

7.2.2.3 String relations 57

7.2.2.4 Logical operator logicalop 58

7.2.3 Pattern match pattern 58

8 Statements 60

8.1 General statement rules 60

8.1.1 Spaces in statements 61

8.1.2 Comment comment 61

8.1.3 Statement argument indirection 62

8.1.4 Post conditional postcond 62

8.1.5 Statement timeout timeout 62

8.1.6 Line reference lineref 63

8.1.6.1 Entry reference entryref 63

8.1.6.2 $TEXT reference textref 64

8.1.6.3 Label reference labelref 64

8.1.6.4 External reference externref 64

8.1.6.5 Library reference libraryref 64

8.1.7 Parameter passing 65

8.1.8 User-defined mnemonicspaces 66

8.2 Statement definitions 67

8.2.1 BREAK 67

8.2.2 CLOSE 68

8.2.3 DO 68

8.2.4 ELSE 70

8.2.5 FOR 70

8.2.6 HALT 72

8.2.7 HANG 72

8.2.8 IF 72

8.2.9 JOB 73

8.2.10 KILL 73

8.2.11 KSUBSCRIPTS 74

8.2.12 KVALUE 75

8.2.13 LOCK 76

8.2.14 MERGE 77

8.2.15 NEW 78

8.2.16 OPEN 79

8.2.17 QUIT 80

8.2.18 READ 81

8.2.19 RLOAD 82

8.2.20 RSAVE 83

8.2.21 SET 83

8.2.22 TCOMMIT 86

8.2.23 TRESTART 87

8.2.24 TROLLBACK 87

8.2.25 TSTART 87

8.2.26 USE 88

8.2.27 VIEW 88

8.2.28 WRITE 89

8.2.29 XECUTE 90

8.2.30 Z 90

8.3 Device Parameters 90

8.3.1 Output timeout 90

9. Character Set Profile charset 91

Section 2: Omega Portability Requirements 93

Introduction 93

1 Character Set 94

2 Expression elements 94

2.1 Names 94

2.2 External routines and names 94

2.3 Local variables 94

2.3.1 Number of local variables 94

2.3.2 Number of subscripts 94

2.3.3 Values of subscripts 94

2.4 Global variables 95

2.4.1 Number of global variables 95

2.4.2 Number of subscripts 95

2.4.3 Values of subscripts 95

2.4.4 Number of nodes 95

2.5 Data types 95

2.6 Number range 95

2.7 Integers 96

2.8 Character strings 96

2.9 Special variables 96

3 Expressions 96

3.1 Nesting of expressions 96

3.2 Results 96

3.3 External References 96

4 Routines and statement lines 96

4.1 Statement lines 96

4.2 Number of statement lines 96

4.3 Number of statements 96

4.4 Labels 97

4.5 Number of labels 97

4.6 Number of routines 97

5 External routine calls 97

6 Character Set Profiles 97

7 Indirection 97

8 Storage space restrictions 97

9 Process-Stack 98

10 Formats 98

10.1 mnemonicspace 98

10.2 controlmnemonic 98

10.3 Parameters 99

11 Transaction processing 99

11.1 Number of modifications in a TRANSACTION 99

11.2 Number of nested TSTARTs within a TRANSACTION 99

12 Other portability requirements 99

Section 3: X3.64 Binding 100

Introduction 100

1 The binding 101

1.1 Control-functions with an effect on $X or $Y 101

1.2 Control-functions with an effect on $KEY 102

1.3 Control-functions with an effect on $DEVICE 102

1.4 Open-ended definitions 102

2 Portability issues 103

2.1 Implementation 104

2.2 Application 104

3 Conformance 104

Annex A: Character Set Profiles (normative) 105

Annex B: Error code translations (informative) 117

Annex C: Metalanguage element dictionary (Informative) 118

Annex D: Embedded SQL (Informative) 120

Annex E: Transportability of Omega Software Systems (informative) 122

Annex F: X3.64 Controlmnemonics (informative) 124

Annex G: charset JIS90 (informative) 125

Annex H: Example Code for Library Functions (informative) 126

Index 139

Introduction

Section 1 consists of nine clauses that describe the Omega extension to the M language. Clause 1 describes the metalanguage used in the remainder of Section 1 for the static syntax. The remaining clauses describe the static syntax and overall semantics of the language. The distinction between "static" and "dynamic" syntax is as follows. The static syntax describes the sequence of characters in a routine as it appears on a tape in routine interchange or on a listing. The dynamic syntax describes the sequence of characters that would be encountered by an interpreter during execution of the routine. (There is no requirement that the Omega extension to M actually be interpreted). The dynamic syntax takes into account transfers of control and values produced by indirection.

1. Scope

This standard describes the Omega extension to the M programming language.

2. Normative References

The following standard(s) contain provisions which, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standard(s) indicated below. Members of ANSI maintain registers of the currently valid standards.

ANSI X3.135-1992 Information Systems - Database Language - SQL

ANSI X3.4-1990 (ASCII Character Set)

ANSI X3.64-1979 R1990 (ANSI Terminal Device Control Mnemonics)

ANSI X11.1-1995

3. Conformance

3.1 Implementations

A *conforming implementation* shall

a) correctly execute all programs conforming to both the Standard and the implementation defined features of the implementation

b) reject all code that contains errors, where such error detection is required by the Standard

c) be accompanied by a document which provides a definition of all implementation-defined features and a conformance statement of the form:

"*xxx* version *v* conforms to X11.7-*yyyy* with the following exceptions:

...

Supported Character Set Profiles are ...

Uniqueness of the values of $SYSTEM is guaranteed by ..."

where the exceptions are those components of the implementation which violate this Standard or for which minimum values are given that are less than those defined in Section 2.

An *MDC conforming implementation* shall be a conforming implementation except that the conforming document shall be this Standard together with any such current MDC documents that the vendor chooses to implement. The conformance statement shall be of the form:

"*xxx* version *v* conforms to X11.7-*yyyy*, as modified by the following MDC documents:

*ddd* (MDC status *m*)

with the following exceptions:

...

Supported Character Set Profiles are ...

Uniqueness of the values of $SYSTEM is guaranteed by ..."

An *MDC strictly conforming implementation* is an MDC conforming implementation whose MDC modification documents only have MDC Type A status and which has no exceptions.

A <*National Body> ... implementation*  is an implementation conforming to one of the above options in which the requirements of Section 2 are replaced by the <National Body> requirements and other extensions required by the <National Body> are implemented.

An implementation may claim more than one level of conformance if it provides a switch by which the user is able to select the conformance level.

3.2 Programs

A *strictly conforming program* shall use only the constructs specified in Section 1 of this standard, shall not exceed the limits and restrictions specified in Section 2 of the Standard and shall not depend on extensions of an implementation or implementation-dependent features.

A *strictly conforming non-ASCII program* is a strictly conforming program, except that the restrictions to the ASCII character set in Section 2 are removed.

A *strictly conforming <National Body> program* is a strictly conforming program, except that the restrictions in Section 2 are replaced by those specified by the <National Body> and any extensions specified by the <National Body> may be used.

A *conforming program* is one that is acceptable to a conforming implementation.

4. Definitions

For the purposes of this standard, the following definitions apply.

4.1 argument (of a statement): Omega statement words are verbs. Their arguments are the objects on which they act.

4.2 array: Omega arrays, unlike those of most other computer languages, are trees of unlimited depth and breadth. Every node may optionally contain a value and may also have zero or more descendant nodes. The name of a subscripted variable refers to the root, and the *n*th subscript refers to a node on the nth level. Arrays vary in size as their nodes are set and killed. See scalar, subscript.

4.3 atom: A singular, most-basic element of a construction. For example, some atoms in an expression are names of variables and functions, numbers, and string literals.

4.4 block, implicit: One or more lines of code within a routine that execute in line as a unit. The argumentless DO statement introduces a block, and each of its lines begins with one or more periods. Blocks may be nested. See level.

**4.5: block, explicit:** One or more statements, comments, and nested blocks that execute in sequence as a unit.

4.6 call by reference: A calling program passes a reference to its actual parameter. If the called subroutine or function changes its formal parameter, the change affects the actual parameter as well. Limited to unsubscripted names of local variables, either scalar or array. See also call by value.

4.7 call by value: A calling program passes the value of its actual parameter to a subroutine or function. Limited to a single value, that is, the value of a scalar variable or of one node in an array. See also call by reference.

4.8 call: A procedural process of transferring execution control to a **callee** by a **caller**.

4.9 callee: The recipient of a **call**.

4.10 caller: The originator of a **call**.

**4.11 character:** (1) A member of a set of elements used for the organization, control, or representation of data. (2) A character is a simple or composite graphic symbol belonging to a conventional set of symbols. There are alphabetic characters, numerical characters (arabic and roman), diacritic characters (for example ˆ ° ̀ ´), punctuation characters (for example . , ; : ! ?), and specific other characters (for example § $ % & { #). Synonyms should be avoided: {graphic, phonetic} symbol, sign, mark, note, cipher.

**4.12 combining character:** A member of an identified subset of the coded character set of ISO/IEC 10646 intended for combination with the preceding non-combining graphic character, or with a sequence of combining characters preceded by a non-combining character (see also *composite sequence*). NOTE - This part of ISO/IEC 10646 specifies several subset collections which include combining characters.

4.13 statement: A statement word (a verb), an optional conditional expression, and zero or more arguments. Statements initiate all actions in M.

**4.14 composite sequence:** A sequence of graphic characters consisting of a non-combining character followed by one or more combining characters (see also *combining characters*). NOTES - 1) A graphic symbol for a composite sequence generally consists of the combination of the graphic symbols of each character in the sequence. 2) A composite sequence is not a character and therefore is not a member of the repertoire of ISO/IEC 10646.

4.15 computationally equivalent: The result of a procedure is the same as if the code provided were executed by a Omega program without error. However, there is no implication that executing the code provided is the method by which the result is achieved.

4.16 concatenation:The act or result of joining two strings together to make one string.

4.17 conditional expression: Guards a statement (sometimes an argument of a statement). Only if the expression's value is true does the statement execute (on the argument). See truthvalue.

4.18 contains: a logical operator that tests whether one string is a substring of another.

4.19 data-cell: in the formal model of Omega execution. It contains the value and subscripts (if any) of a variable, but not the name of the variable. Many variable names may point to a data-cell due to parameters passed by reference. See also name-table, value-table.

**4.20 default state:** The state that is assumed when no state has been explicitly specified.

4.21 descriptor: uniquely defines an element. It comprises various characteristics of the element that distinguish the element from all other similar elements.

4.22 device-dependent: That which depends on the device in question.

**4.23 diacritic:** Character which is not a [letter] of the latin alphabet and which is placed over, under, or through a letter or combination of letters indicating a semantic or phonetic value different from that given the unmarked or otherwise marked letter. A letter with a diacritic is a composite character. NOTE - The point of german umlaut"-character should be regarded as diacritic. [also called a *diacritical mark*]

**4.24 diacritic mark:** An attribute used of a character applied to denote a variation of a letter.

**4.25 digit:** A graphic character used to represent the numeric value, or part thereof, of a number. Examples: decimal digits, hexadecimal digits.

4.26 empty: an entity that contains nothing. For example, an empty string contains no characters; it exists but has zero length. See also null string, NULL character.

4.27 environment: a set of distinct names. For example, in one global environment all global variables have distinct names. Similar to a directory in many operating systems.

4.28 evaluate: to derive a value.

4.29 execute: to perform the operations specified by the statements of the language.

**4.30 “executing” a namevalue:** a namevalue is “executed” when it is used in an indirect reference (i.e., @Ref) or subscripted indirectness (i.e., @Ref@(3)).

4.31 extract: to retrieve part of a value, typically contiguous characters from a string.

4.32 extrinsic: a function or variable defined and created by Omega code, distinct from the primitive functions or special variables of the language. See intrinsic.

4.33 follow: to come after according to some ordering sequence. See also sorts after.

4.34 function: a value-producing subroutine whose value is determined by its arguments. Intrinsic functions are defined elements of the language, while extrinsic functions are programmed in M.

4.35 global variable: a scalar or array variable that is public, available to more than one job, and persistent, outliving the job. See local variable.

4.36 graphic: a visible character (as opposed to most control characters).

**4.37 graphic character:** A character, other than a control function, that has a visual representation normally handwritten, printed, or displayed.

4.38 hidden: unseen. The NEW statement hides local variables. Also pertains to unseen elements invoked to define the operation of some statements and functions.

4.39 intrinsic: a primitive function or variable defined by the language standard as opposed to one defined by Omega code. See extrinsic.

4.40 job: A single operating system process running a Omega program.

4.41 label: Identifies a line of code.

**4.42 letter:** (1) A letter (or alphabetic character) is a character that is an individual unspecific basic unit of the latin alphabet, irrespective of the shape and any graphical realization on a medium. A letter can be specified as a *small letter* or *capital letter*. (2) A graphic character used for writing natural language, normally representing a sound of the language.

4.43 library: a collection of library elements, with unique names, which are referenced using a single library name. A library is defined as being either mandatory or optional.

4.44 library element: an individual function that is separately defined and accessible from an Omega process using the library reference syntax.

**4.45 ligature:** (1) A composite character joining two or more letters. There are ligatures that are conventionalized units of a national variant of the latin alphabet, and ligatures that are caused by the font used in a document. Maybe the first ones should be named *ligature characters*, the last ones *ligature font elements*. [Language dependent. Only *ligature characters* are taken into consideration] (2) Two or more letters written together. The resulting symbol is in some cases considered equivalent with the originating letters, in some cases it is considered a separate entity.

4.46 local variable: A scalar or array variable that is private to one job, not available to other jobs, and disappears when the job terminates. See global variable.

4.47 lock: To claim or obtain exclusive access to a resource.

4.48 Omega Standard Library: all libraries and library elements defined within the Omega Standard, whether mandatory or optional.

4.49 mapping:The logical association or substitution of one element for another.

4.50 map: The act of mapping.

4.51 metalanguage: Underlined terms used in the formal description of the Omega language.

4.52 modulo: An arithmetic operator that produces the remainder after division of one operand by another. There are many interpretations of how this operation is performed in the general computing field. Omega explicitly defines the result of this computation.

4.53 multidimensional: Used in reference to arrays to indicate that the array can have more than one dimension.

4.54 naked: A shorthand reference to one level of the tree forming a global array variable. The full reference is defined dynamically.

4.55 name-table: In the formal model of Omega execution, a set of variable names and their pointers to data-cells.

**4.56 negative:** A numeric value less than zero. Zero is not negative.

4.57 node: One element of the tree forming an array. It may have a value and it may have descendants.

4.58 NULL character: The character that is internally coded as code number 0 (zero). A string may contain any number of occurrences of this character (up to the maximum string length). A string consisting of one NULL character has a length of 1 (one).

4.59 null string: 1. A string consisting of 1 (one) NULL character; 2. A string consisting of 0 (zero) characters.

4.60 object: An entity considered as a whole in relation to other entities.

**4.61 ordering:** Bringing strings of characters into a well-defined dequence using a string comparison specification.

4.62 own:To have exclusive access to a resource. In Omega this pertains to devices.

4.63 parameter: A qualifier of a statement modifies its behavior (for example by imposing a time out), or augments its argument (for example by setting characteristics of a device). Some parameters are expressions, and some have the form keyword=value. See argument.

4.64 parameter (of a function or subroutine): The calling program provides actual parameters. In the called function or subroutine, formal parameters relate by position to the caller's actual arguments. See also call by reference, call by value, parameter passing.

4.65 parameter passing: This alliterative phrase refers to the association of actual parameters with formal parameters when calling a subroutine or function.

4.66 partition:The random access memory in which a job runs.

4.67 piece: A part of a string, a sub-string delimited by chosen characters.

4.68 pointer:Indirection allows one Omega variable to refer, or point to, another variable or the argument of a statement.

4.69 portable: Omega code that conforms to the portability section of the standard.

**4.70 positive:** A numeric value greater than zero. Zero is not positive.

4.71 post-conditional: See conditional expression.

4.72 primitives: The basic elements of the language.

4.73 process-stack: In the formal model of Omega execution, a push-down stack that controls the execution flow and scope of variables.

4.74 relational:Pertaining to operators that compare the values of their operands.

4.75 scalar:Single-valued, without descendants. See array.

4.76 scope (of a statement): The range of other statements affected by the statement, as in loop control, block structure, and conditional execution.

4.77 scope (of a local variable):The range of statements for which the variable is visible, from its creation to its deletion, or from its appearance in a NEW statement to the end of the subroutine, function, or block. Scope is not textual, but dynamic, controlled by the flow of execution.

4.78 sorts after: To come after according to an ordering sequence that is based on a collating algorithm. See also follows.

4.79 subscript: An expression whose value specifies one node of an array. Its value may be an integer, a floating point number, or any string. Subscripts are sparse, that is, only those that have been defined appear in the array. See array, scalar.

4.80 trails: “*A* trails *B*” means that (“ ”\_*A*) ]] (“ ”\_*B*) in the appropriate collation sequence; if not specified, it refers to the sequence used for local variables.

4.81 truthvalue:The value of an expression considered as a number. Non-zero is true, and zero is false.

4.82 tuple: A sequence of a predetermined number of descriptors (usually a name and a series of subscripts) that identifies a member of a set.

4.83 type: Omega recognizes only one data type, the string of variable length. Arithmetic operations interpret strings as numbers, and logical operations further interpret the numbers as true or false. See also truthvalue.

4.84 unbound: In the formal model of Omega execution, the disassociation of a variable's name from its value.

4.85 undefined: Pertaining to a variable that is not visible to a statement.

4.86 unsubscripted: See scalar.

4.87 value-denoting: Representing or having a value.

4.88 value-table: In the formal model of Omega execution, a set of data-cells.

4.89 variable: Omega variables may be local or global, scalar or array.

5. Metalanguage Description

The primitives of the metalanguage are the ASCII characters. The metalanguage operators are defined as follows:

Operator Meaning

::= definition

[ ] option

| | grouping

... optional indefinite repetition

L list

V value

SP space

VB vertical bar

The following visible representations of ASCII characters required in the defined syntactic objects are used: SP (space), CR (carriage-return), LF (line-feed), FF (form-feed), and VB (vertical bar). Also, where necessary to avoid confusion with the “option” metalanguage operator, OB is used to represent the open bracket character ( [ ) and CB is used to represent the close bracket character ( ] ).

In general, defined syntactic objects will have designators which are underlined names spelled with lower case letters, e.g., name, expr, etc. Concatenation of syntactic objects is expressed by horizontal juxtaposition, choice is expressed by vertical juxtaposition. The ::= symbol denotes a syntactic definition. An optional element is enclosed in square brackets [ ], and three dots ... denote that the previous element is optionally repeated any number of times. The definition of name, for example, is written:

┌─ ─┐

name ::= │ % │ │ digit │ ...

│ ident │ │ ident │

└─ ─┘

The vertical bars are used to group elements or to make a choice of elements more readable.

Special care is taken to avoid any danger of confusing the square brackets in the metalanguage with the ASCII graphics ] and [. Normally, the square brackets will stand for the metalanguage symbols.

The unary metalanguage operator L denotes a list of one or more occurrences of the syntactic object immediately to its right, with one comma between each pair of occurrences. Thus,

L name is equivalent to [ WS ] name [ WS ] [ , [ WS ] name [ WS ] ] ... .

The binary metalanguage operator V places the constraint on the syntactic object to its left that it must have a value which satisfies the syntax of the syntactic object to its right. For example, one might define the syntax of a hypothetical EXAMPLE statement with its argument list by

examplestatement ::= EXAMPLE WS L exampleargument

where

│ expr │

exampleargument ::= │ │

│ @ expratom V L exampleargument │

This example states: after evaluation of indirection, the statement argument list consists of any number of exprs separated by commas. In the static syntax (i.e., prior to evaluation of indirection), occurrences of @ expratom may stand in place of nonoverlapping sublists of statement arguments. Usually, the text accompanying a syntax description incorporating indirection will describe the syntax after all occurrences of indirection have been evaluated.

6. Routine routine

The routine is a string made up of the following symbols:

The graphic, including the space character represented as SP, and also,

the carriage-return character represented as CR,

the line-feed character represented as LF,

the form-feed character represented as FF.

Each routine begins with its routinehead, which contains the identifying routinename. The routinehead is followed by the routinebody, which contains the code to be executed. The routinehead is not part of the executed code.

routine ::= routinehead routinebody

6.1 Routine head routinehead

routinehead ::= routinename eol

routinename ::= name

┌─ ─┐

name ::= │ % │ │ digit │ ...

│ ident │ │ ident │

└─ ─┘

control ::= The ASCII/M codes 0-31 and 127 (see Annex A for the definition of ASCII/M)

digit ::= The ASCII/M codes 48-57 (characters '0' - '9')

graphic ::= Those characters in the current charset which are not control characters.

ident ::= The ASCII/M codes 65-90 and 97-122 ('A'-'Z' and 'a'-'z') are ident characters, all other characters in the range 0-127 are not ident characters. Additional characters, with codes greater than 127, may be defined as ident through the algorithm specified in ^$CHARACTER(charsetexpr,"IDENT")

eol ::= CR LF

names differing only in the use of corresponding upper and lower case letters are not equivalent.

6.2 Routine body routinebody

The routinebody is a sequence of methods terminated by an eor.

routinebody ::= method ... eor

eor ::= CR FF

6.2.1 Method method

The method is the primary callable unit of code and the primary unit of variable scoping, and consists of a callable label, optional formallist, and the block (called a method block) that implements the method.

method ::= label [ formallist ] WS block

formallist ::= ( [ L name ] )

WS ::= │ SP │ ...

│ eol [ dots ] │

dots ::= [ spaces ] | . [ spaces ] | ... spaces

spaces ::= SP ...

**6.2.2 Label label**

Each occurrence of a label that starts a method is called a *defining occurrence* of label. An error occurs with ecode = "M57" if there are two or more defining occurrences of label with the same spelling in one routinebody.

label ::= │ name │

│ intlit │

**6.2.3 Block block**

The block is the primary unit for delimiting the scope of control flow, and consists of an optional sequence of statements, comments, and nested blocks and extblocks. These codeitems are separated by one or more spaces and eol characters. However, whenever two back to back codeitems within a block are both statements, there *must* be a cs between them. A block encountered as a codeitem within another block is called a nested block. A block's scope includes all codeitems within, only the statements outside of nested blocks are said to fall within the block's immediate scope.

block ::= OB code CB

code ::= [ WS ] codeitem [ WS codeitem ] ... [ WS ]

│ block │

codeitem ::= │ statement [ cs ] │

│ extblock │

│ comment │

cs ::= WS ; WS

comment ::= { [ graphic ] ... }

The use of the extblock form is allowed only within the context of an embedded Omega program (see 6.4 Embedded programs).

**6.2.4 External block extblock**

extblock ::= & extid OB extcode CB

extcode ::= [ graphic ] ...

extid ::= │ SQL │

│ Z[unspecified]│

An extblock is a block that contains code conforming to the syntax of a programming language or standard other than Omega; the block's extid prefix identifies the language or standard.

The exact syntax of the extcode is defined by the external programming language or standard. In the case of extid being SQL this standard is X3.135 (see also Annex D). extids differing only in the use of corresponding upper and lower case letters are equivalent. extids not beginning with the letter Z are reserved for future extension of the language.

Note: An external block implies that one or more Omega methods or routines may be created by some compilation process, replacing any external syntax with appropriate Omega statement lines, function calls etc. An external block or external block pre-processor does not, therefore, need to adhere to the portability requirements of Section 2 although the equivalent Omega methods, routines, and implementation should.

**6.2.5 Line line**

line ::= graphic ... eol

A routine can also be viewed as divided into lines, in which each line contains all the characters between two eol characters along with the second of the two eol characters. Although this view of a routine has no bearing on routine execution, it is used by language elements that need to access or refer to the pieces of a routine, such as the RLOAD and RSAVE statements, and the $STACK and $TEXT functions.

**6.3 Routine execution**

Routines are executed in a sequence of blocks. Block processing may be altered by the initiation of transactions or the occurrence of error conditions.

**6.3.1 Block processing**

A block is invoked either as a method block, the dependent block of an executing statement, or as a nested block:

a) A method block is invoked when its method is invoked by the instance of a DO block, a JOB statement (in the new process), a doargument, an exfunc, or an exvar. The method executes by processing passed parameters (See section 8.1.7 Parameter Passing), then invoking an implicit exclusive NEW on all local variables except for those passed in as parameters, then executing its method block. Execution of the method ends when its method block terminates.

b) A dependent block is invoked under conditions created by the execution of the statement to which it belongs, and the results of the block's termination are also so constrained.

c) A nested block is invoked when encountered during the execution flow of the block that contains it (the *outer* block). When it terminates, execution of the outer block resumes with the next statement after the CB of the nested block.

Within a given block execution proceeds sequentially, beginning at the leftmost statement and proceeding left to right, and from the top down. Routine flow statements DO, FOR, IF, QUIT, TRESTART, XECUTE, exfunc extrinsic functions, and exvar extrinsic variables, provide exception to this execution flow. (See also 6.3.4 Error Processing.) In general, each statement's argument is evaluated in a left-to-right order, except as explicitly noted elsewhere in this document.

The block ends when 1) the CB that ends the block is encountered, thus terminating this block; 2) any of the conditions of a blockless IF in the immediate scope of the block evaluates to 0; or 3) when a QUIT statement is encountered. The effects of executing a QUIT depends upon the type of block being terminated:

a) When a QUIT terminates a method block, its method also terminates. The results of terminating a method depend upon how the method was invoked, and are described in the relevant sections.

b) When a QUIT terminates a dependent block, its statement also terminates. The results of terminating a statement depend on the statement, and are described in the relevant sections.

c) When a QUIT terminates a nested block, its outer block also terminates. If the outer block is also a nested block, repeat step c; otherwise, go to step a or b depending on the block type.

(see 8.2.16 for a description of the actions of QUIT). (See also the DO statement).

**6.3.2 Transaction processing**

A TRANSACTION is the execution of a sequence of statements that begins with a TSTART and ends with either a TCOMMIT or a TROLLBACK, and that is not within the scope of any other TRANSACTION. A TRANSACTION may be restartable, serializable, or both, depending on parameters specified in the TSTART that initiates the TRANSACTION. (See 8.2.24 TSTART.) These properties affect execution of the TRANSACTION as described below.

TSTART adds one to the intrinsic special variable $TLEVEL, which is initialized to zero when a process begins execution. TCOMMIT subtracts one from $TLEVEL if $TLEVEL is greater than zero. TROLLBACK sets $TLEVEL to zero. A process is within a TRANSACTION whenever its $TLEVEL value is greater than zero. A process is not within a TRANSACTION whenever its $TLEVEL value is zero.

If, as a result of a TCOMMIT, $TLEVEL would become zero, an attempt is made to COMMIT the TRANSACTION. A COMMIT causes the global variable modifications made within the TRANSACTION to become durable and accessible to other processes.

A ROLLBACK is performed if, within a TRANSACTION, either a TROLLBACK or a HALT statement is executed. A ROLLBACK rescinds all global variable modifications performed within the scope of the TRANSACTION, removes any nrefs from the LOCK-LIST that were not included in the LOCK-LIST when the TRANSACTION started (i.e. when $TLEVEL changed from zero to one), and removes any RESTART CONTEXT-STRUCTUREs for both the TRANSACTION linked list and the PROCESS-STACK linked list, discarding the CONTEXT-STRUCTUREs. Omega errors do not cause an implicit ROLLBACK. (See the LOCK statement for definitions of nref and LOCK-LIST.)

Global variable modifications carried out by statements executed within a TRANSACTION are subject to the following rules:

a) A process that is outside of a TRANSACTION cannot access the global variable modifications made within a TRANSACTION until that TRANSACTION has been COMMITted.

b) A process that is inside a TRANSACTION is not explicitly excluded from accessing modifications made by other processes. However, a process cannot COMMIT a TRANSACTION that has accessed the global variable modifications of any other uncommitted TRANSACTION before that other TRANSACTION has been committed.

c) If the transparameters within the argument to the TSTART initiating the TRANSACTION specifies serializability, then all global modifications performed by the TRANSACTION and all other concurrently executing TRANSACTIONs must be equivalent to some serial, non-overlapping execution of those TRANSACTIONs.

If it has been determined that a TRANSACTION in progress either cannot or is unlikely to conform to the above-stated rules, then the TRANSACTION implicitly RESTARTs. In addition, the TRESTART statement explicitly causes the TRANSACTION to RESTART.

The actions of a RESTART depend on whether it is restartable. A TRANSACTION is restartable if the initiating TSTART specifies a restartargument. (See 8.2.24 TSTART.) A RESTART of a restartable TRANSACTION causes execution to resume with the initial TSTART. A RESTART of a non-restartable TRANSACTION ends in an error (ecode="M27").

The following discussion uses terms defined in the Variable Handling (see 7.1.2.2) and Process-Stack (see 7.1.2.3) models and, like those subclauses, does not imply a required implementation technique.

Execution of a RESTART occurs as follows:

a) The frame at the top of the PROCESS-STACK is examined. If the frame's linked list of CONTEXT-STRUCTUREs contains entries, they are processed in last-in-first-out order from their creation. If the CONTEXT-STRUCTURE is exclusive, all entries in the currently active local variable NAME-TABLE are pointed to empty DATA-CELLs. In all cases, the CONTEXT-STRUCTURE NAME-TABLEs are copied to the currently active NAME-TABLEs. For each RESTART CONTEXT-STRUCTURE, $TLEVEL is decremented by one until $TLEVEL reaches 0 (zero) or the list is exhausted. If $TLEVEL does not reach 0 (zero), then:

1) if the frame contains formallist information, it is processed as described by step d in the description of the QUIT statement (see 8.2.16).

2) the frame is removed and step a repeats.

b) The naked indicator is restored from the CONTEXT-STRUCTURE that triggered $TLEVEL to reach 0 (zero).

c) A ROLLBACK is performed. If the TRANSACTION is not restartable, RESTART terminates and an error condition occurs with ecode= "M27"

d) $TRESTART is incremented by 1. RESTART terminates and execution continues with the initial TSTART, which includes re-evaluating postcond, if any, and tstartargument, if any.

**6.3.3 Error processing**

Error trapping provides a mechanism by which a process can execute specifiable statements in the event that $ECODE becomes non-empty. The following facilities are provided:

The $ETRAP special variable may be set to either the empty string or to code to be invoked when $ECODE becomes non-empty. Stacking of the contents of $ETRAP is performed via the NEW statement.

$ECODE provides information describing existing error conditions. $ECODE is a comma-surrounded list of conditions.

The $STACK function and $STACK variable provide stack related information.

$ESTACK counts stack levels since $ESTACK was last NEWed.

An Error Processing transfer of control effects the immediate termination of all FOR statements in the current execution environment whose scope includes the current statement. Without changing the PROCESS-STACK, and without provision for a return of control, it transfers execution control to special error processing code. This code is implicitly incorporated into the current execution environment immediately preceding the next statement in the normal execution sequence, and execution continues at the left of the special code. For purposes of this transfer each statement argument is considered to have its own statementword (see 8.1 General statement rules). The special code consists of the following nested block, where *x* is the value of $ETRAP:

OB *x* QUIT:$QUIT "" QUIT CB

An Error Processing transfer of control is performed when:

a) The value of $ECODE changes from an empty string to some other value as the result of an error or a SET statement.

b) $ECODE is not the empty string and a QUIT statement removes a PROCESS-STACK level at which $STACK($STACK,"ECODE") would return a non-empty string, and, at the new PROCESS-STACK level, $STACK($STACK,"ECODE") would return an empty string (in other words, when a QUIT takes the process from a frame in which an error occurred to a frame where no error has occurred).

When $STACK($STACK,"ECODE") returns a non-empty string and the value of $ECODE changes to a non-empty string, the following actions are performed:

a) It associates the $STACK information about the failure as if it were associated with the frame identified by $STACK+1.

b) It transfers control to the following special error processing code, which is implicitly incorporated into the current execution environment immediately preceding the next statement in the normal execution sequence. The special code consists of the following block, as follows:

OB TROLLBACK:$TLEVEL QUIT:$QUIT "" QUIT CB

**7. Expression expr**

The expression, expr, is the syntactic element which denotes the execution of a value-producing calculation. Expressions are made up of expression atoms separated by binary, string, arithmetic, or truth-valued operators.

expr ::= expratom [ exprtail ] ...

**7.1 Expression atom expratom**

The expression atom, expratom, is the basic value-denoting object of which expressions are built.

expratom ::= │ glvn │

│ expritem │

**7.1.1 Variables**

The Omega standard uses the terms *local variables* and *global variables* somewhat differently from their connotation in certain other computer languages. This subclause provides a definition of these terms as used in the Omega environment.

A Omega routine, or set of routines, runs in the context of an operating system process. During its execution, the routine will create and modify variables that are restricted to its process. It can also access (or create) variables that can be shared with other processes. At the termination of the process, the process-specific variables cease to exist. The variables created for shared use persist beyond the life of the process, and they may be accessed by other processes.

M uses the term *local variable* to denote variables that are created for use during a single process activation. These variables are not available to other processes. Within a process they are only visible within the method that created them, and any called methods to which they are passed. Omega does include certain constructs, the NEW statement and parameter passing, which can further limit the availability of certain variables to specific routines or parts of routines.

A *global variable* is one that is created by a process, but is permanent and shared. As soon as a process creates, modifies or deletes a global variable outside of a TRANSACTION, other processes accessing that global variable outside of a TRANSACTION receive its modified form. (See 6.3.1 Transaction processing for a definition of TRANSACTION and information on how TRANSACTIONs affect global modifications.) Global variables do not disappear when a process terminates. Like local variables, global variables are available to all routines executed within a process.

M has no explicit declaration or definition statements. Local and global variables, both non-subscripted and subscripted, are automatically created as data is stored into them, and their data contents can be referred to once information has been stored. Since the language has only one data type - string - there is no need for type declarations or explicit data type conversions. Array structures can be multidimensional with data simultaneously stored at all levels including the variable name level. Subscripts can be positive, negative, or zero; they can be integer or noninteger numbers as well as nonnumeric strings (other than empty strings).

**7.1.2 Variable name glvn**

The metalanguage element glvn is defined so as to be satisfied by the syntax of gvn, lvn, or ssvn.

│ lvn │

glvn ::= │ gvn │

│ ssvn │

**7.1.2.1 Local variable name lvn**

lvn ::= │ rlvn │

│ @ expratom V lvn │

rlvn ::= │ name [ ( L expr ) ] │

│ @ lnamind @ ( L expr ) │

lnamind ::= rexpratom V lvn

│ rlvn │

rexpratom ::= │ rgvn │

│ rssvn │

│ expritem │

See 7.1.2.4 for the definition of rgvn. See 7.1.4 for the definition of expritem.

A local variable name is either unsubscripted or subscripted; if it is subscripted, any number of subscripts separated by commas is permitted. An unsubscripted occurrence of lvn may carry a different value from any subscripted occurrence of lvn.

When lnamind is present it is always a component of an rlvn. If the value of the rlvn is a subscripted form of lvn, then some of its subscripts may have originated in the lnamind. In this case, the subscripts contributed by the lnamind appear as the first subscripts in the value of the resulting rlvn, separated by a comma from the (non-empty) list of subscripts appearing in the rest of the rlvn.

**7.1.2.2 Local variable handling**

In general, the operation of the local variable symbol table can be viewed as follows. Prior to the initial setting of information into a variable, the data value of that variable is said to be undefined. Data is stored into a variable with statements such as SET, READ, or FOR. Subsequent references to that variable return the data value that was most recently stored. When a variable is killed, as with the KILL statement, that variable and all of its array descendants (if any) are deleted, and their data values become undefined.

No explicit syntax is needed for a routine or subroutine to have access to the local variables of its caller. Except when the NEW statement or parameter passing is being used, a subroutine or called routine (the callee) has the same set of variable values as its caller and, upon completion of the called routine or subroutine, the caller resumes execution with the same set of variable values as the callee had at its completion.

The NEW statement provides scoping of local variables. It causes the current values of a specified set of variables to be saved. The variables are then set to undefined data values. Upon returning to the caller of the current routine or subroutine, the saved values, including any undefined states, are restored to those variables. Parameter passing, including the DO statement, extrinsic functions, and extrinsic variables, allows parameters to be passed into a subroutine or routine without the callee being concerned with the variable names used by the caller for the data being passed or returned.

The formal association of local variables with their values can best be described by a conceptual model. This model is NOT meant to imply an implementation technique for a Omega implementation.

The value of a variable may be described by a relationship between two structures: the NAME-TABLE and the VALUE-TABLE. (In reality, at least two such table sets are required, one pair per executing process for process-specific local variables and one pair for system-wide global variables.) Since the value association process is the same for both types of variables, and since issues of scoping due to parameter passing or nested environments apply only to local variables, the discussion that follows will address only local variable value association. It should be noted, however, that while the overall structures of the table sets are the same, there are two major differences in the way the sets are used. First, the global variable tables are shared. This means that any operations on the global tables, e.g., SET or KILL, by one process, affect the tables for all processes. Second, since scoping issues of parameter passing and the NEW statement are not applicable to global variables, there is always a one-to-one relationship between entries in the global NAME-TABLE (variable names) and entries in the global VALUE-TABLE (values).

The NAME-TABLE consists of a set of entries, each of which contains a name and a pointer. This pointer represents a correspondence between that name and exactly one DATA-CELL from the VALUE-TABLE. The VALUE-TABLE consists of a set of DATA-CELLs, each of which contains zero or more tuples of varying degrees. The degree of a tuple is the number (possibly 0) of elements or subscripts in the tuple list. Each tuple present in the DATA-CELL has an associated data value.

The NAME-TABLE entries contain every non-subscripted variable or array name (name) known, or accessible, by the process in the current environment. The VALUE-TABLE DATA-CELLs contain the set of tuples that represent all variables currently having data-values for the process. Every name (entry) in the NAME-TABLE refers (points) to exactly one DATA-CELL, and every entry contains a unique name. Several NAME-TABLE entries (names) can refer to the same DATA-CELL, however, and thus there is a many-to-one relationship between (all) NAME-TABLE entries and DATA-CELLs. A name is said to be *bound* to its corresponding DATA-CELL through the pointer in the NAME-TABLE entry. Thus the pointer is used to represent the correspondence and the phrase *change the pointer* is the equivalent to saying *change the correspondence so that a name now corresponds to a possible different DATA-CELL (value)*. NAME-TABLE entries are also placed in the PROCESS-STACK (see 7.1.2.3 Process-Stack).

The value of an unsubscripted lvn corresponds to the tuple of degree 0 found in the DATA-CELL that is bound to the NAME-TABLE entry containing the name of the lvn. The value of a subscripted lvn (array node) of degree n also corresponds to a tuple in the DATA-CELL that is bound to the NAME-TABLE entry containing the name of the lvn. The specific tuple in that DATA-CELL is the tuple of degree n such that each subscript of the lvn has the same value as the corresponding element of the tuple. If the designated tuple doesn't exist in the DATA-CELL then the corresponding lvn is said to be *undefined*.

In the following figure, the variables and array nodes have the designated data values.

*VAR1* = "Hello"

*VAR2* = 12.34

*VAR3* = "abc"

*VAR3*("Smith","John",1234)=123

*VAR3*("Widget","red") = -56

Also, the variable *DEF* existed at one time but no longer has any data or array value, and the variable *XYZ* has been bound through parameter passing to the same data and array information as the variable *VAR2*.

NAME-TABLE VALUE-TABLE DATA-CELLS

┌──────────┐

*VAR1*----------> │()="Hello"│

└──────────┘

┌─────────┐

*VAR2*----------> │()=12.34 │

*XYZ*-----------> └─────────┘

┌──────────────────────────┐

*VAR3*----------> │()="abc" │

│("Smith","John",1234)=123 │

│("Widget","red")=-56 │

└──────────────────────────┘

┌─────────────────┐

*DEF*-----------> │ │

└─────────────────┘

The initial state of a process prior to execution of any Omega code consists of an empty NAME-TABLE and VALUE-TABLE. When information is to be stored (set, given, or assigned) into a variable (lvn):

a) If the name of the lvn does not already appear in an entry in the NAME-TABLE, an entry is added to the NAME-TABLE which contains the name and a pointer to a new (empty) DATA-CELL. The corresponding DATA-CELL is added to the VALUE-TABLE without any initial tuples.

b) Otherwise, the pointer in the NAME-TABLE entry which contained the name of the lvn is extracted. The operations in steps c and d refer to tuples in that DATA-CELL referred to by this pointer.

c) If the lvn is unsubscripted, then the tuple of degree 0 in the DATA-CELL has its data value replaced by the new data value. If that tuple did not already exist, it is created with the new data value.

d) If the lvn is subscripted, then the tuple of subscripts in the DATA-CELL (i.e., the tuple created by dropping the name of the lvn; the degree of the tuple equals the number of subscripts) has its data value replaced by the new data value. If that tuple did not already exist, it is created with the new data value.

When information is to be retrieved, if the name of the lvn is not found in the NAME-TABLE, or if its corresponding DATA-CELL tuple does not exist, then the data value is said to be undefined. Otherwise, the data value exists and is retrieved. A data value of the empty string (a string of zero length) is not the same as an undefined data value.

When a variable is deleted (killed):

a) If the name of the lvn is not found in the NAME-TABLE, no further action is taken.

b) If the lvn is unsubscripted, all of the tuples in the corresponding DATA-CELL are deleted.

c) If the lvn is subscripted, let *N* be the degree of the subscript tuple formed by removing the name from the lvn. All tuples that satisfy the following two conditions are deleted from the corresponding DATA-CELL:

1) The degree of the tuple must be greater than or equal to *N*, and

2) The first *N* arguments of the tuple must equal the corresponding subscripts of the lvn.

In this formal language model, even if all of the tuples in a DATA-CELL are deleted, neither the DATA-CELL nor the corresponding names in the NAME-TABLE are ever deleted. Their continued existence is frequently required as a result of parameter passing and the NEW statement.

**7.1.2.3 Process-Stack**

The PROCESS-STACK is a virtual last-in-first-out (LIFO) list (a simple push-down stack) used to describe the behavior of Omega. It is used as an aid in describing how Omega appears to work and does not imply that an implementation is required to use such a stack to achieve the specified behavior. Two types of items, or frames, will be placed on the PROCESS-STACK: block frames (including DOs, XECUTEs, exfunc, and exvar) and error frames (for errors that occur during error processing):

a) Block frames contain the execution location of the doargument or xargument. The execution location of a process is a descriptor of the location of the statement and possible argument currently being executed. This descriptor includes, at minimum, the routinename and the character position following the current statement or argument.

c) Error frames contain information about error conditions during error processing (see 6.3.2 Error processing).

The term CONTEXT-STRUCTURE is used to refer to a set of information related to the maintenance of the process context.

**7.1.2.4 Global variable name gvn**

│ rgvn │

gvn ::= │ │

│ @ expratom V gvn │

│ ^( L expr ) │

rgvn ::= │ ^ [ VB environment VB ] name [ ( L expr ) ] │

│ @ gnamind @ ( L expr ) │

gnamind ::= rexpratom V gvn

environment ::= expr

The prefix ^ uniquely denotes a global variable name. A global variable name is either unsubscripted or subscripted; if it is subscripted, any number of subscripts separated by commas is permitted. An abbreviated form of subscripted gvn is permitted, called the *naked reference*, in which the prefix is present but the environment, name and an initial (possibly empty) sequence of subscripts is absent but implied by the value of the *naked indicator*. An unsubscripted occurrence of gvn may carry a different value from any subscripted occurrence of gvn.

When environment is present it identifies a specific set of all possible names.

When gnamind is present it is always a component of an rgvn. If the value of the rgvn is a subscripted form of gvn, then some of its subscripts may have originated in the gnamind. In this case, the subscripts contributed by the gnamind appear as the first subscripts in the value of the resulting rgvn, separated by a comma from the (non-empty) list of subscripts appearing in the rest of the rgvn.

Every executed occurrence of gvn affects the naked indicator as follows. If, for any positive integer *m*, the gvn has the nonnaked form

*N*(*v1* , *v2* , ... , *vm* )

then the *m*-tuple *N*, *v1* , *v2* , ... , *vm−1* , is placed into the naked indicator when the gvn reference is made. A subsequent naked reference of the form

^(*s1* , *s2* , ... , *si* ) (*i* positive)

results in a global reference of the form

*N*(*v1* , *v2* , ... , *vm−1* , *s1* , *s2* , ... , *si* )

after which the *m+i−1*-tuple *N* , *v1* , *v2* , ... , *si−1* is placed into the naked indicator. Prior to the first executed occurrence of a nonnaked form of gvn, the value of the naked indicator is undefined. A nonnaked reference without subscripts or a ROLLBACK, or a change of the default global environment leaves the naked indicator undefined. When a gvn is encountered in the form of a naked reference and the naked indicator is undefined, an error condition occurs with ecode="M1".

The effect on the naked indicator described above occurs regardless of the context in which gvn is found; in particular, an assignment of a value to a global variable with the statement SET gvn = expr does not affect the value of the naked indicator until after the right-side expr has been evaluated. The effect on the naked indicator of any gvn within the right-side expr will precede the effect on the naked indicator of the left-side gvn.

**7.1.3 Structured system variable ssvn**

│ rssvn │

ssvn ::= │ │

│ @ expratom V ssvn │

│ ^$ [ VB environment VB ] ssvname [ ( L expr ) ] │

rssvn ::= │ │

│ @ ssvnamind @ ( L expr ) │

ssvnamind ::= rexpratom V ssvn

The prefix ^$ uniquely denotes a structured system variable name. The parenthesized list of exprs following the ssvname are called subscripts; a ssvn may be either subscripted or unsubscripted; if it is subscripted, any number of subscripts separated by commas is permitted (the allowed values and/or interpretation of each subscript is defined for each individual ssvname). Structured system variable names (ssvnames) differing only in the use of corresponding upper and lower case letters are equivalent.

When ssvnamind is present it is always the component of a rssvn. If the value of the rssvn is a subscripted form of ssvn, then some of its subscripts may have originated in the ssvnamind. In this case, the subscripts contributed by the ssvnamind appear as the first subscripts in the value of the resulting rssvn, separated by a comma from the (non-empty) list of subscripts appearing in the rest of the rssvn.

Values may not be assigned to ssvns and ssvns may not be KILLed unless the semantics of these operations are explicitly defined. The environment form of the ssvn syntax may only refer to the default environment unless the ssvn is explicitly defined to permit the use of environments other than the default. A reference to such an ssvn which refers to an environment that is not explicitly permitted is erroneous and causes an error condition with ecode = "M59". Other references to ssvns using the environment syntax however, due to technical reasons or security concerns, may be restricted by implementors to a restricted set of possible environments. An attempt to violate this restriction causes an error condition with an implementor-specified ecode beginning with "Z".

The meaning of the individual subscripts of a ssvn is explicitly defined for each ssvn. The standard contains the following ssvnames:

│ C[HARACTER] │

│ D[EVICE] │

│ G[LOBAL] │

ssvname ::= │ J[OB] │

│ LIBRARY │

│ L[OCK] │

│ R[OUTINE] │

│ S[YSTEM] │

│ Z[unspecified] │

Unused structured system variable names beginning with an initial letter other than Z are reserved for future enhancement of the standard.

**7.1.3.1 ^$CHARACTER**

^$C[HARACTER] ( charsetexpr )

charsetexpr ::= expr V charset

^$CHARACTER provides information regarding the available Character Set Profiles on a system, such as collation order and pattern code definitions.

When and only when a Character Set Profile identified by charset exist, ^$CHARACTER(charset) has a value; all nonempty string values are reserved for future extension of the standard.

Data manipulation and the execution of statements within a process are performed in the context of the process charset. (See 7.1.3.4 ^$JOB)

Input-Transformation:

^$CHARACTER( charsetexpr1 , expr V "INPUT" , charsetexpr2 ) = expr V algoref

│ emptystring │

│ $$ labelref │

algoref ::= │ $& externref │

│ $ functionname │

emptystring ::= a string of zero length.

This node specifies the input-transformation algorithm which is performed on a string in the process Character Set Profile charset1 when it is retrieved from a global or routine which uses charset2 or transmitted from a device using charset2. The algoref specifies the algorithm by which this translation is accomplished, if no input-transformation algorithm is defined, an empty-string value is used. The conversion of the string *old* to the string *new* using the input-transformation algorithm *transform* may be evaluated by executing: ("S *new*="\_*transform*\_"(*old*)").

Output-Transformation:

^$CHARACTER( charsetexpr1 , expr V "OUTPUT" , charsetexpr2 ) = expr V algoref

This node specifies the output-transformation algorithm which is performed on a string in the process Character Set Profile charset1 when it is stored in a global or routine which uses charset2 or transmitted to a device using charset2. The algoref specifies the algorithm by which this translation is accomplished, if no output-transformation algorithm is defined, an empty-string value is used. The conversion of the string *old* to the string *new* using the output-transformation algorithm *transform* may be evaluated by executing: ("S *new*="\_*transform*\_"(*old*)").

Valid name characters:

^$CHARACTER( charsetexpr , expr V "IDENT" ) = expr V algoref

This node specifies the identification algorithm used to determine which characters in a charset are valid for use in names (i.e. is a character in the set ident).

The ident truth-value *truth*, of a character *char* using an identification algorithm *ident*, may be evaluated by executing the expression: ("S *truth*="\_*ident*\_"($ASCII(*char*))"). When *truth* is "true", *char* is an ident; when *truth* is "false", *char* is not an ident. Note that for $ASCII(*char*) values less than 128, 65-90 and 97-122 are required to be "true" and all other values less than 128 are required to be "false". If the identification algorithm node is undefined, or is the empty string, then it will return "false" for all $ASCII(*char*) greater than 127; values less than 128 will be returned as indicated.

patcode definition:

^$CHARACTER( charsetexpr , expr V "PATCODE" , expr V patcode ) = expr V algoref

This node identifies the pattern testing algorithm that determines which characters of charset match the specified patcode; if this node is not defined, or is the empty string, then no characters in the charset will match that patcode. The patcode truth-value *truth* of a character *char* using a nonempty-string pattern testing algorithm *pattest* may be evaluated by executing the expression: ("S *truth*="\_*pattest*\_"($ASCII(*char*))"). When *truth* is "true", *char* belongs to the specified patcode; when *truth* is "false", *char* does not belong to that patcode.

Collation Algorithm:

^$CHARACTER( charsetexpr , expr V "COLLATE" ) = expr V algoref

This node identifies the collation algorithm for the specified Character Set Profile ( charset ).

**7.1.3.2 ^$DEVICE**

^$D[EVICE] ( devicexpr )

devicexpr ::= expr V device

device ::= devicespecifier; an implementation specific device identifier.

^$DEVICE provides information about the existence, operational characteristics and availability of devices.

Note: The holding of information about a device when it is not open may be transitory. There are also likely to be more devices in a system which could be opened by a Omega process than will have information stored in ^$DEVICE.

Device characteristic information for a device is stored beneath the ^$DEVICE(devicexpr) node:

^$DEVICE( devicexpr , expr V "CHARACTER") = charsetexpr

This node identifies the current Character Set Profile of the specified device. The Character Set Profile is assigned to the device in an implementation-specific manner.

^$DEVICE ( devicexpr , expr V deviceattribute )

This contains the primary value or values associated with this deviceattribute. Additional values may be stored in descendants of this node.

When a device is opened then values for the deviceattributes are created in ^$DEVICE. These may be retained after the device is closed. The range of deviceattribute names and the format of the values is defined by the mnemonicspace in use for the device.

^$DEVICE ( devicexpr , expr V "MNEMONICSPACE" ) = mnemonicspace

This node identifies the mnemonicspace currently in effect for the device. If there is no mnemonicspace in effect then this node has the value of the empty string.

^$DEVICE ( devicexpr , expr V "MNEMONICSPEC" , expr V mnemonicspace ) = emptystring

This node identifies a mnemonicspace that has been associated with the device through the OPEN and USE statements. All nonempty string values are reserved for future extension of the standard.

When the mnemonicspace in use for the device defines an output timeout as described in 8.3.1, it shall also define the following two members of ^$DEVICE:

a) the value of ^$DEVICE ( devicexpr , expr V “OUTTIMEOUT” ) shall equal the value of the most recently executed OUTTIMEOUT deviceparam for the device. It shall equal 0 when no OUTTIMEOUT deviceparam has executed for the device.

b) the value of ^$DEVICE ( devicexpr , expr V “OUTSTALLED” ) shall indicate the output timeout status of the device. If the most recently executed output-producing argument of a READ or WRITE statement timed out, then this value shall be 1. If that argument did not time out, this value shall be 0.

**7.1.3.3 ^$GLOBAL**

^$G[LOBAL] ( gvnexpr )

gvnexpr ::= expr V name

^$GLOBAL provides information about the existence and characteristics of globals.

When and only when a global identified by gvnexpr exists, ^$GLOBAL(gvnexpr) has a value; all nonempty string values are reserved for future extension of the standard. Global characteristic information is stored beneath the ^$GLOBAL(gvnexpr) node:

^$GLOBAL( gvnexpr , expr V "CHARACTER") = charsetexpr

This node identifies the Character Set Profile of the specified global. When the first node in a global is created, and the node ^$GLOBAL(gvnexpr,"CHARACTER") has a $DATA value of zero, the value assigned is that of ^$JOB($JOB,"CHARACTER"). The result of killing a gvn does not alter the characteristics stored in ^$GLOBAL for that gvn.

Collation Algorithm:

^$GLOBAL( gvnexpr , expr V "COLLATE" ) = expr V algoref

This node identifies the collation algorithm to be used when collation is required for a reference to this global. The collation value *order* for a subscript-string *subscript*, and a collation algorithm *collate* may be determined by executing the expression: ("S *order*="\_*collate*\_"(*subscript*)"). In all cases a collation algorithm must return a distinct *order* for each distinct *subscript*.

When the first node of a global *global* is created, and the collation algorithm node ^$GLOBAL(*"global"*,"COLLATE") has a $DATA value of zero, then the value of the current process' Character Set Profile collation algorithm ( $GET(^$CHARACTER(^$JOB($JOB,"CHARACTER"),"COLLATE")) ) is assigned as the global's collation algorithm ( ^$GLOBAL("*global"*,"COLLATE") ).

**7.1.3.4 ^$JOB**

^$J[OB] ( processid )

processid ::= expr V jobnumber

^$JOB provides information about the existence and characteristics of processes in a system.

When and only when a process identified by processid exists, ^$JOB(processid) has a value; all nonempty string values are reserved for future enhancement of the standard. Process characteristics are stored beneath the ^$JOB(processid) node.

**7.1.3.4.1 Characteristic: Character Set Profile**

^$JOB( processid , expr V "CHARACTER") = charsetexpr

This node identifies the active Character Set Profile in use by the process indicated by processid. Unless otherwise modified via the processparameters of the JOB statement, when a process is created ^$JOB($JOB,"CHARACTER") is set to the charset of the process that created it.

**7.1.3.4.2 Characteristic: Available Function Libraries**

^$JOB( processid , expr1 V "LIBRARY" , expr2 ) = libraryexpr

This node identifies a library currently available to the process. The order in which the librarys are searched to locate a specific libraryelement is defined by the collating order of the values of expr2 for the specified librarys.

**7.1.3.4.3 Characteristic: Devices**

^$JOB( processid , expr V "$PRINCIPAL" ) = devicexpr (principal device)

^$JOB( processid , expr V "$IO" ) = devicexpr (current device)

^$JOB( processid , expr V "OPEN" , devicexpr ) = (for each OPENed device)

These nodes specify the device information associated with process processid. The node "$PRINCIPAL" is the value of $PRINCIPAL for that process. The node "$IO" is the value of $IO (current device being used) for that process. The devicexpr nodes beneath "OPEN" are the device identifiers for all the devices which are currently OPENed for that process.

7.1.3.5 ^$LIBRARY

^$LIBRARY ( libraryexpr )

libraryexpr ::= expr V library

^$LIBRARY provides information about the availability of libraries and library elements in a system.

When and only when a library *l* exists, ^$LIBRARY( *l* ) has a value; all non-empty string values are reserved for future expansion of the standard. Library information is stored beneath the ^$LIBRARY(library) node:

^$LIBRARY ( libraryexpr , expr V "ELEMENT" , libraryelementexpr )

libraryelementexpr ::= expr V libraryelement

When and only when a library *l* and libraryelement *e* exist, ^$LIBRARY( *l* , "LIBRARY" , *e* ) has a value; all non-empty string values are reserved for future expansion of the standard.

**7.1.3.6 ^$LOCK**

^$L[OCK] ( expr V nref )

will provide information on the existence and operational characteristics of locked names.

**7.1.3.7 ^$ROUTINE**

^$R[OUTINE] ( routinexpr )

routinexpr ::= expr V routinename

^$ROUTINE provides information about the existence and characteristics of routines.

When and only when a routine identified by routinexpr exists, ^$ROUTINE(routinexpr) has a value; all nonempty string values are reserved for future enhancement of the standard. Process characteristics are stored beneath the ^$ROUTINE(routinexpr) node:

^$ROUTINE( routinexpr , expr V "CHARACTER" ) = charsetexpr

This node identifies the Character Set Profile in which routine routinexpr is stored.

When a routine is created and ^$ROUTINE(routinexpr,"CHARACTER") for that routine has a $DATA value of zero, then this node is assigned the current value of the node ^$JOB($JOB,"CHARACTER").

**7.1.3.8 ^$SYSTEM**

^$S[YSTEM] ( systemexpr )

systemexpr ::= expr V system

system ::= syntax of $SYSTEM intrinsic special variable

^$SYSTEM provides information about the characteristics of systems. A system represents the domain of concurrent processes for which $JOB is unique; the current system is identified by the svn $SYSTEM. The second level subscripts of ^$SYSTEM not beginning with the letter "Z" are reserved for future enhancement of the standard.

System Character Set Profile:

^$SYSTEM( systemexpr , expr V "CHARACTER" ) = charsetexpr

This node specifies the charset which the specified system uses for interpretation of all system-wide name values (syntactic elements, e.g. ssvn names, statementwords, svn names, etc). Note that this allows an implementation to provide $Z[\*] names, etc which include idents other than those in ASCII/M.

System Collation Algorithm

^$SYSTEM( systemexpr , expr V "COLLATE" ) = expr V algoref

This node identifies the collation algorithm which the specified system uses for determining collation order for system syntactic elements.

**7.1.3.9 ^$Z[unspecified]**

^$Z[unspecified] ( unspecified )

will provide implementation-specific information. Z is the initial letter for defining non-standard structured system variables. The requirement that ^$Z be used permits the unused initial letters to be reserved for future enhancement of the standard without altering the execution of existing programs which observe the rules of the standard.

**7.1.3.10 ssvns specifying default environments**

The following ssvns, specifying default environments, are defined. This clause pertains to the following four ssvns:

^$JOB(processid,"DEVICE") default device environment

^$JOB(processid,"GLOBAL") default global environment

^$JOB(processid,"JOB") default JOB environment

^$JOB(processid,"LOCK") default lock environment

^$JOB(processid,"ROUTINE") default routine environment

A process may always obtain and assign a value to these nodes, where processid = $JOB. However, for technical reasons or security concerns, implementations may restrict access to these nodes for processids other than the current processid. An attempt to violate this restriction causes an error condition with an implementor-specified ecode beginning with "Z".

When a process starts, the values of these ssvns are in general defined by the implementation. However, a process initiated by a JOB statement inherits the default environments of the initiating process, unless explicitly specified in the jobargument.

Explicit qualification of a labelref, routineref, gvn, nref, or devn with an environment overrides the default environment for that one reference.

Assigning a non-existent environment to one of these ssvns is not in itself erroneous. However, an attempt to refer to a routine, global, lock, or device in the non-existent environment causes an error condition with an ecode = "M26".

**7.1.4 Expression item expritem**

│ strlit │

│ numlit │

│ exfunc │

│ exvar │

expritem ::= │ svn │

│ function │

│ unaryop expratom │

│ ( expr ) │

**7.1.4.1 String literal strlit**

┌─ ─┐

│ "" │

strlit ::= " │ │ ... "

│ nonquote │

└─ ─┘

nonquote ::= any of the characters in graphic except the quote character.

In words, a string literal is bounded by quotes and contains any string of printable characters, except that when quotes occur inside the string literal, they occur in adjacent pairs. Each such adjacent quote pair denotes a single quote in the value denoted by strlit, whereas any other printable character between the bounding quotes denotes itself. An empty string is denoted by exactly two quotes.

**7.1.4.2 Numeric literal numlit**

The integer literal syntax, intlit, which is a nonempty string of digits, is defined here.

intlit ::= digit ...

The numeric literal numlit is defined as follows.

numlit ::= mant [ exp ]

mant ::= │ intlit [ . intlit] │

│ . intlit │

┌─ ─┐

exp ::= E │ + │ intlit

│ - │

└─ ─┘

The value of the string denoted by an occurrence of numlit is defined in the following two subclauses.

**7.1.4.3 Numeric data values**

All variables, local, global, and special, have values which are either defined or undefined. If defined, the values may always be thought of and operated upon as strings. The set of numeric values is a subset of the set of all data values.

Only numbers which may be represented with a finite number of decimal digits are representable as numeric values. A data value has the form of a number if it satisfies the following restrictions.

a) It shall contain only digits and the characters "−" and ".".

b) At least one digit must be present.

c) "." occurs at most once.

d) The number zero is represented by the one-character string "0".

e) The representation of each positive number contains no "−".

f) The representation of each negative number contains the character "−" followed by the representation of the positive number which is the absolute value of the negative number. (Thus, the following restrictions describe positive numbers only.)

g) The representation of each positive integer contains only digits and no leading zero.

h) The representation of each positive number less than 1 consists of a "." followed by a nonempty digit string with no trailing zero. (This is called a *fraction*.)

i) The representation of each positive non-integer greater than 1 consists of the representation of a positive integer (called the *integer part* of the number) followed by a fraction (called the *fraction part* of the number).

Note that the mapping between representable numbers and representations is one-to-one. An important result of this is that string equality of numeric values is a necessary and sufficient condition of numeric equality.

**7.1.4.4 Meaning of numlit**

Note that numlit denotes only nonnegative values. The process of converting the spelling of an occurrence of numlit into its numeric data value consists of the following steps.

a) If the mant has no ".", place one at its right end.

b) If the exp is absent, skip step c.

c) If the exp has a plus or has no sign, move the "." a number of decimal digit positions to the right in the mant equal to the value of the intlit of exp, appending zeros to the right of the mant as necessary. If the exp has a minus sign, move the "." a number of decimal digit positions to the left in the mant equal to the value of the intlit of exp, appending zeros to the left of the mant as necessary.

d) Delete the exp and any leading or trailing zeros of the mant.

e) If the rightmost character is ".", remove it.

f) If the result is empty, make it "0".

**7.1.4.5 Numeric interpretation of data**

Certain operations, such as arithmetic, deal with the numeric interpretations of their operands. The numeric interpretation is a mapping from the set of all data values into the set of all numeric values, described by the following algorithm. Note that the numeric interpretation maps numeric values into themselves.

(Note: The *head* of a string is defined to be a substring which contains an identical sequence of characters in the string to the left of a given point and none of the characters in the string to the right of that point. A head may be empty or it may be the entire string.)

Consider the argument to be the string *S*.

First, apply the following sign reduction rules to *S* as many times as possible, in any order.

a) If *S* is of the form + *T*, then remove the +. (Shorthand: + *T* ⇒ *T*)

b) − + *T* ⇒ − *T*

c) −− *T* ⇒ *T*

Second, apply one of the following, as appropriate.

a) If the leftmost character of *S* is not "−", form the longest head of *S* which satisfies the syntax description of numlit. Then apply the algorithm of 7.1.4.4 to the result.

b) If *S* is of the form − *T*, apply step a) above to *T* and append a "−" to the left of the result. If the result is "−0", change it to "0".

The *numeric expression* numexpr is defined to have the same syntax as expr. Its presence in a syntax description serves to indicate that the numeric interpretation of its value is to be taken when it is executed.

numexpr ::= expr

**7.1.4.6 Integer interpretation**

Certain functions deal with the integer interpretations of their arguments. The integer interpretation is a mapping from the set of all data values onto the set of all integer values, described by the following algorithm.

First, take the numeric interpretation of the argument. Then remove the fraction, if present. If the result is empty or "−", change it to "0".

The *integer expression* intexpr is defined to have the same syntax as expr. Its presence in a syntax definition serves to indicate that the integer interpretation of its value is to be taken when it is executed.

intexpr ::= expr

**7.1.4.7 Truth-value interpretation**

The truth-value interpretation is a mapping from the set of all data values onto the two integer values 0 (false) and 1 (true), described by the following algorithm. Take the numeric interpretation. If the result is not "0", make it "1".

The *truth-value expression* tvexpr is defined to have the same syntax as expr. Its presence in a syntax definition serves to indicate that the truth-value interpretation of its value is to be taken when it is executed.

tvexpr :: = expr

**7.1.4.8 Extrinsic function exfunc**

╷ $ labelref ╷

exfunc ::= $ │ libraryref │ actuallist

│ externref │

Extrinsic functions invoke a subroutine to return a value. When an extrinsic function is executed, the current execution level and the current execution location are saved in a block frame on the PROCESS-STACK. The actuallist parameters are then processed as described in 8.1.7.

Execution continues either in the specified externref or at the first statement of the method specified by the labelref. This method must contain a formallist in which the number of names is greater than or equal to the number of names in the actuallist, otherwise an error occurs with ecode = "M58".

Upon return from the subroutine the value of the argument of the QUIT statement that terminated the subroutine is returned as the value of the exfunc.

**7.1.4.9 Extrinsic variable exvar**

│ $ labelref │

exvar ::= $ │ libraryref │

│ externref │

An extrinsic special variable whose labelref is *x* is identical to the extrinsic function:

$$x()

Note that label *x* must have a (possibly empty) formallist.

**7.1.4.10 Intrinsic special variable names svn**

Intrinsic special variables are denoted by the prefix $ followed by one of a designated list of names. Intrinsic special variable names differing only in the use of corresponding upper and lower case letters are equivalent. The standard contains the following intrinsic special variable names:

D[EVICE]

EC[ODE]

ES[TACK]

ET[RAP]

H[OROLOG]

I[O]

IOR[EFERENCE]

J[OB]

K[EY]

PIOR[EFERENCE]

P[RINCIPAL]

Q[UIT]

R[EFERENCE]

ST[ACK]

S[TORAGE]

SY[STEM]

T[EST]

TL[EVEL]

TR[ESTART]

X

Y

Z[unspecified]

Unused intrinsic special variable names beginning with an initial letter other than Z are reserved for future enhancement of the standard.

The formal definition of the syntax of svn is a choice from among all of the individual svn syntax definitions of this subclause.

│ syntax of $DEVICE intrinsic special variable │

│ syntax of $IO intrinsic special variable │

svn ::= │ . │

│ . │

│ . │

│ syntax of $Y intrinsic special variable │

│ syntax of $Z[unspecified] intrinsic special variable │

Any implementation of the language must be able to recognize both the abbreviation and the full spelling of each intrinsic special variable name.

7.1.4.10.1 $DEVICE

$D[EVICE]

reflects the status of the current device. If the status of the device does not reflect any error-condition, the value of $DEVICE, when interpreted as a truth-value, will be 0 (false). If the status of the device would reflect any error-condition, the value of $DEVICE, when interpreted as a truth-value, will be 1 (true). When the process is initiated $DEVICE is given the value of the empty string if $IO is given a value which is the empty string, otherwise it is given an implementation-dependent value.

$DEVICE will give status code and meaning in one access. Its value is one of

│ M │

│ M,I │

│ M,I,T │

╵ ╵

where M is an MDC defined value , I is an implementor defined value and T is explanatory text.

The value of M, when interpreted as a truth value, will be equal to 0 (zero) when no significant change of status is being reported. Any nonzero value indicates a significant change of status.

The value of I is an implementation-specific value for the relevant status-information.

The value of T is implementation specific.

Note: Since M, I, and T are separated by commas, the values of M and I cannot contain this character.

7.1.4.10.2 $ECODE

$EC[ODE]

contains information about an error condition. This information is loaded by the implementation after detecting an erroneous condition, or by the application via the SET statement. When the value of $ECODE is the empty string, normal routine execution rules are in effect. When $ECODE contains anything else, the execution rules in 6.3.2 (Error processing) are active. When a process is initiated, but before any statements are processed, the value of $ECODE is the empty string.

The syntax of a non-empty value returned by $ECODE is as follows:

, │ ecode , │ ...

│ M │

ecode ::= │ U │ [ noncomma ... ]

│ Z │

noncomma ::= any of the characters in graphic except the comma

character.

Note: ecodes beginning with:

M are reserved for the MDC

U are reserved for the user

Z are reserved for the implementation

All other values are reserved.

7.1.4.10.3 $ESTACK

$ES[TACK]

counts stack levels in the same way as $STACK, however, a NEW $ESTACK saves the value of $ESTACK and then assigns $ESTACK the value of 0. When a process is initiated, but before any statements are processed, the value of $ESTACK is 0 (zero).

7.1.4.10.4 $ETRAP

$ET[RAP]

contains code which is invoked in the event an error condition occurs. See 6.3.2- Error processing. When a process is initiated, but before any statements are processed, the value of $ETRAP is the empty string.

The value of $ETRAP may be stacked with the NEW statement; NEW $ETRAP has the effect of saving the current instantiation of $ETRAP and creating a new instantiation initialized with the same value.

The value of $ETRAP is changed with the SET statement. Changing the value of $ETRAP with a SET statement instantiates a new trap; it does not save the old trap.

A QUIT from $ETRAP, either explicit or implicit (i.e., SET $ETRAP="DO ^ETRAP" has an implicit QUIT at its end with an empty argument, if appropriate) will function as if a QUIT had been issued at the "current" $STACK. Behavior at the "popped" level will be determined by the value of $ECODE. If $ECODE is empty, execution proceeds normally. Otherwise, $ETRAP is invoked at the new level.

7.1.4.10.5 $HOROLOG

$H[OROLOG]

gives date and time with one access. Its value is *D* , *S* where *D* is an integer value counting days since an origin specified below, and *S* is an integer value modulo 86,400 counting seconds. The value of $HOROLOG for the first second of December 31, 1840 is defined to be 0,0. *S* increases by 1 each second and *S* clears to 0 with a carry into *D* on the tick of midnight.

7.1.4.10.6 $IO

$I[O]

identifies the current I/O device (see 8.2.2 and 8.2.25). Its value has the form of expr. When the process is initiated $IO is given the value of $PRINCIPAL if an implicit OPEN and USE for the device specified by $PRINCIPAL is executed by the implementation. If the implementation does not execute this OPEN and USE then $IO is given the value of the empty string.

7.1.4.10.7 $IOREFERENCE

$IOR[EFERENCE]

identifies the current I/O device (see 8.2.2 and 8.2.25). Its value has the syntax of devn with the following restrictions:

a) When the process is initiated $IOREFERENCE is given the value of $PRINCIPAL if an implicit OPEN and USE for the device specified by $PRINCIPAL is executed by the implementation. If the implementation does not execute this OPEN and USE then $IOREFERENCE is given the value of the empty string.

b) If the last statement that changed $IOREFERENCE included an environment, then the value returned by $IOREFERENCE shall include that environment; otherwise the value of $IOREFERENCE shall not include environment.

c) An environment whose value has the form of a number as defined in 7.1.4.3 appears as a numlit, spelled as its numeric interpretation.

d) An environment whose value does not have the form of a number as defined in 7.1.4.3 appears as a strlit.

7.1.4.10.8 $JOB

$J[OB]

Each executing process has its own job number, a positive integer which is the value of $JOB. The job number of each process is unique to that process within a domain of concurrent processes defined by the implementor. $JOB is constant throughout the active life of a process.

7.1.4.10.9 $KEY

$K[EY]

contains the control-sequence which terminated the last READ statement from the current device (including any introducing and terminating characters). If no READ statement was issued to the current device or when no terminator was used, the value of $KEY will be the empty string. The effect of a READ \*glvn on $KEY is unspecified. When the process is initiated $KEY is given the value of the empty string if $IO is given a value which is the empty string, otherwise it is given an implementation-dependent value.

If a Character Set Profile input-transform is in effect, then this is also applied to the value stored in $KEY. Certain mnemonicspaces may also specify that $KEY contains values as a result of other I/O statements.

See (READ statement) and (WRITE statement).

7.1.4.10.10 $PIOREFERENCE

$PIOR[EFERENCE]

identifies the principal I/O device. When the process is initiated, $PIOREFERENCE is given the value of $PRINCIPAL with the following restrictions:

a) If $PRINCIPAL is the empty string, then $PIOREFERENCE is the empty string.

b) If $PRINCIPAL is not the empty string, then $PIOREFERENCE shall include an environment.

7.1.4.10.11 $PRINCIPAL

$P[RINCIPAL]

identifies the principal I/O device, which is defined in the following fashion:

a. If the process is initiated by another Omega process then $PRINCIPAL is given the value of $PRINCIPAL of the initiating process, unless overriden by implementation-specific JOB parameters.

b. If the process is initiated from a specific device then $PRINCIPAL is given the identifier of the device.

c. Otherwise $PRINCIPAL is given an implementation-specific value.

$PRINCIPAL is constant throughout the active life of a process.

7.1.4.10.12 $QUIT

$Q[UIT]

returns 1 if the current PROCESS-STACK frame was invoked by an exfunc or exvar, and therefore a QUIT would require an argument. Otherwise, $QUIT returns 0 (zero). When a process is initiated, but before any statements are processed, the value of $QUIT is 0 (zero).

7.1.4.10.13 $REFERENCE

$R[EFERENCE]

returns the namevalue of the most recently referenced gvn, on which the current value of the naked indicator is based; for the behavior after a reference to the function $QUERY see 7.1.5.15. When the process is initiated $REFERENCE is given a value which is the empty string.

The value of $REFERENCE may be set to either the empty string, or to a namevalue, indicating a gvn. A side-effect of setting $REFERENCE equal to the empty string is that the naked indicator will become undefined. A side-effect of setting $REFERENCE to a namevalue is that the naked indicator will change as if the indicated gvn had been referenced.

7.1.4.10.14 $STACK

$ST[ACK]

gives the current level of the PROCESS-STACK. $STACK contains an integer value of zero or greater. When a process is initiated, but before any statements are processed, the value of $STACK is 0 (zero). See 7.1.2.3 (process-stack) for a description of stack behavior.

7.1.4.10.15 $SYSTEM

$SY[STEM]

Each implementation must return a value in $SYSTEM which represents uniquely the system representing the domain of concurrent processes for which $JOB is unique. Its value is *V,S* where *V* is an integer value allocated by the MDC to an implementor and *S* is defined by that implementor in such a way as to be able to be unique for all the implementor's systems.

7.1.4.10.16 $TLEVEL

$TL[EVEL]

indicates whether a TRANSACTION is currently in progress. When the process is initiated $TLEVEL is given the value 0. TSTART adds 1 to $TLEVEL. When $TLEVEL is greater than 0, TCOMMIT subtracts 1 from $TLEVEL. A ROLLBACK or RESTART sets $TLEVEL to 0.

7.1.4.10.17 $TRESTART

$TR[ESTART]

indicates how many RESTARTs have occurred since the initiation of a TRANSACTION. When the process is initiated $TRESTART is given the value 0, and it is set to 0 by the successful completion of TCOMMIT or TROLLBACK. Each RESTART adds 1 to $TRESTART.

7.1.4.10.18 $X

$X

has a nonnegative integer value which approximates the value of the horizontal co-ordinate of the active position on the current device. It is initialized to zero by any control-function or format that involves a move to the start of a line. When the process is initiated $X is given the value 0 if $IO is given a value which is the empty string, otherwise it is given an implementation-dependent value.

The unit in which $X is expressed is initially equal to 'characters'. Certain formats may change this.

When any control-function would leave the cursor in a position so that the horizontal co-ordinate would be uncertain, the value of $X will not be changed. In such cases the value of $DEVICE will be an error-code.

If a Character Set Profile input-transform is in effect, then $X is modified in accordance with the input prior to any transform taking place. If a Character Set Profile output-transform is in effect, then $X is modified in accordance with the output after any transform takes place.

See 8.2.17 (READ statement) 8.2.25 (USE statement) and 8.2.27 (WRITE statement).

7.1.4.10.19 $Y

$Y

has a nonnegative integer value which approximates the value of the vertical co-ordinate of the active position on the current device. It is initialized to zero by any control-function or format that involves a move to the start of a page. When the process is initiated $Y is given the value 0 if $IO is given a value which is the empty string, otherwise it is given an implementation-dependent value.

The unit in which $Y is expressed is initially equal to 'lines'. Certain formats may change this.

When any control-function would leave the cursor in a position so that the vertical co-ordinate would be uncertain, the value of $Y will not be changed. In such cases, the value of $DEVICE will be an error-code.

If a Character Set Profile input-transform is in effect, then $Y is modified in accordance with the input prior to any transform taking place. If a Character Set Profile output-transform is in effect, then $Y is modified in accordance with the output after any transform takes place.

See 8.2.17 (READ statement) 8.2.25 (USE statement) and 8.2.27 (WRITE statement).

7.1.4.10.20 $Z

$Z[unspecified]

Z is the initial letter reserved for defining non-standard intrinsic special variables. The requirement that $Z be used permits the unused initial letters to be reserved for future enhancement of the standard without altering the execution of existing routines which observe the rules of the standard.

**7.1.4.11 Unary operator unaryop**

│ ' │ *(Note: apostrophe)*

unaryop ::= │ + │

│ - │ *(Note: hyphen)*

There are three unary operators: ' (not), + (plus), and − (minus).

Not inverts the truth value of the expratom immediately to its right. The value of 'expratom is 1 if the truth-value interpretation of expratom is 0; otherwise its value is 0. Note that '' performs the truth-value interpretation.

Plus is merely an explicit means of taking a numeric interpretation. The value of +expratom is the numeric interpretation of the value of expratom.

Minus negates the numeric interpretation of expratom. The value of −expratom is the numeric interpretation of −*N*, where *N* is the value of expratom.

Note that the order of application of unary operators is right-to-left.

**7.1.4.12 Name value namevalue**

namevalue ::= expr

A namevalue has the syntax of a glvn with the following restrictions:

a) The glvn is not a naked reference.

b) Each subscript whose value has the form of a number appears as specified in 7.1.4.3.

c) Each subscript whose value does not have the form of a number as defined in 7.1.4.3 appears as a sublit, defined as follows:

sublit ::= " │ "" │ "

│ subnonquote │...

where subnonquote is defined as follows:

subnonquote ::= any character valid in a subscript, excluding the quote symbol.

d) The environment appears as defined in b. and c. for subscripts.

**7.1.5 Intrinsic function function**

Intrinsic functions are denoted by the prefix $ followed by one of a designated list of names, followed by a parenthesized argument list. Intrinsic function names differing only in the use of corresponding upper and lower case letters are equivalent. The following function names are defined:

╷ A[SCII] │

│ C[HAR] │

│ D[ATA] │

│ E[XTRACT] │

│ F[IND] │

│ FN[UMBER] │

│ G[ET] │

│ J[USTIFY] │

│ L[ENGTH] │

│ NA[ME] │

functionname ::= │ O[RDER] │

│ P[IECE] │

│ QL[ENGTH] │

│ QS[UBSCRIPT] │

│ Q[UERY] │

│ R[ANDOM] │

│ RE[VERSE] │

│ S[ELECT] │

│ ST[ACK] │

│ T[EXT] │

│ TR[ANSLATE] │

│ V[IEW] │

│ Z[unspecified] │

Unused function names beginning with an initial letter other than Z are reserved for future enhancement of the standard.

The formal definition of the syntax of function is a choice from among all of the individual function syntax definitions in this subclause.

│ syntax of $ASCII function │

│ syntax of $CHAR function │

│ . │

function ::= │ . │

│ . │

│ syntax of $VIEW function │

│ syntax of $Z[unspecified] function │

Any implementation of the language must be able to recognize both the abbreviation and the full spelling of each function name.

**7.1.5.1 $ASCII**

$A[SCII] ( expr )

This form produces an integer value as follows:

a) −1 if the value of expr is the empty string.

b) Otherwise, an integer *n* associated with the leftmost character of the value of expr, such that $ASCII($CHAR(*n*)) = *n*.

$A[SCII] ( expr , intexpr )

This form is similar to $ASCII(expr) except that it works with the intexprth character of expr instead of the first. Formally, $ASCII(expr,intexpr) is defined to be $ASCII($EXTRACT(expr,intexpr)).

**7.1.5.2 $CHAR**

$C[HAR] ( L intexpr )

This form returns a string whose length is the number of argument expressions which have nonnegative values. Each intexpr in the closed interval [0,127] maps into the ASCII character whose code is the value of intexpr; this mapping is order-preserving. Each negative-valued intexpr maps into no character in the value of $CHAR. Each intexpr greater than 127 maps into a character in a manner defined by the current charset of the process.

**7.1.5.3 $DATA**

$D[ATA] ( glvn )

This form returns a nonnegative integer which is a characterization of the glvn. The value of the integer is *p*+*d*, where:

*d* = 1 if the glvn has a defined value, i.e., the NAME-TABLE entry for the name of the glvn exists, and the subscript tuple of the glvn has a corresponding entry in the associated DATA-CELL; otherwise, *d*=0.

*p* = 10 if the variable has descendants; i.e., there exists at least one tuple in the glvn's DATA-CELL which satisfies the following conditions:

a) The degree of the tuple is greater than the degree of the glvn, and

b) the first *N* arguments of the tuple are equal to the corresponding subscripts of the glvn where *N* is the number of subscripts in the glvn.

If no NAME-TABLE entry for the glvn exists, or no such tuple exists in the associated DATA-CELL, then *p*=0.

**7.1.5.4 $EXTRACT**

$E[XTRACT] ( expr )

This form returns the first (leftmost) character of the value of expr. If the value of expr is the empty string, the empty string is returned.

$E[XTRACT] ( expr , intexpr )

Let *s* be the value of expr, and let *m* be the integer value of intexpr. $EXTRACT(*s*,*m*) returns the *m*th character of *s*. If *m* is less than 1 or greater than $LENGTH(*s*), the value of $EXTRACT is the empty string. (1 corresponds to the leftmost character of *s*; $LENGTH(*s*) corresponds to the rightmost character.)

$E[XTRACT] ( expr , intexpr1 , intexpr2 )

Let *n* be the integer value of intexpr2. $EXTRACT(*s*,*m*,*n*) returns the string between positions *m* and *n* of *s*. The following cases are defined:

a) *m* > *n*. Then the value of $E is the empty string.

b) *m* = *n*. $E(*s*,*m*,*n*) = $E(*s*,*m*).

c) *m* < *n* '> $L(*s*).

$E(*s*,*m*,*n*) = $E(*s*,*m*) concatenated with $E(*s*,*m*+1,*n*).

That is, using the concatenation operator \_ of 7.2.1.1, $E(*s*,*m*,*n*) = $E(*s*,*m*)\_$E(*s*,*m*+1)\_...\_$E(*s*,*m*+(*n*−*m*)).

d) *m* < *n* and $L(*s*) < *n*.

$E(*s*,*m*,*n*) = $E(*s*,*m*,$L(*s*)).

**7.1.5.5 $FIND**

$F[IND] ( expr1 , expr2 )

This form searches for the leftmost occurrence of the value of expr2 in the value of expr1. If none is found, $FIND returns zero. If one is found, the value returned is the integer representing the number of the character position immediately to the right of the rightmost character of the found occurrence of expr2 in expr1. In particular, if the value of expr2 is empty, $FIND returns 1.

$F[IND] ( expr1 , expr2 , intexpr )

Let *a* be the value of expr1, let *b* be the value of expr2, and let *m* be the value of intexpr. $FIND(*a*,*b*,*m*) searches for the leftmost occurrence of *b* in *a*, beginning the search at the max(*m*,1) position of *a*. Let *p* be the value of the result of $FIND($EXTRACT(*a*,*m*,$LENGTH(*a*)),*b*). If no instance of *b* is found (i.e., *p*=0), $FIND returns the value 0; otherwise, $FIND(*a*,*b*,*m*) = *p* + max(*m*,1) − 1.

**7.1.5.6 $FNUMBER**

$FN[UMBER] ( numexpr , fncodexpr )

fncodexpr ::= expr V fncode

fncode ::= [ fncodatom ... ]

│ fncodp │

│ fncodt │

fncodatom ::= │ , │ *(note, comma)*

│ + │

│ - │ *(note, hyphen)*

╷ ╷

fncodp ::= │ P │

│ p │

╵ ╵

╷ ╷

fncodt ::= │ T │

│ t │

╵ ╵

This form shall return a value that is the value of numexpr edited by applying each fncodatom according to the following rules. The order of application is not significant:

fncodatom Action

fncodp Represent negative numexpr values in parentheses. Let *A* be the absolute value of numexpr. Use of fncodp will result in the following:

1) If numexpr < 0, the result will be "("\_*A*\_")".

2) If numexpr '< 0, the result will be " "\_*A*\_" ".

fncodt Represent numexpr with a trailing rather than a leading "+" or "−" sign. Note: if sign suppression is in force (either by default on positive values, or by design using the "−" fncodatom), use of fncodt will result in a trailing space character.

, Insert comma delimiters every third position to the left of the decimal (present or assumed) within numexpr. Note: no comma shall be inserted which would result in a leading comma character.

+ Force a plus sign ("+") on positive values of numexpr. Position of the "+" (leading or trailing) is dependent on whether or not fncodt is present.

− Suppress the negative sign "−" on negative values of numexpr.

All other values for fncodatom are reserved. Note: Zero is neither positive nor negative.

If fncodexpr equals an empty string, no special formatting is performed and the result of the expression is the original value of numexpr.

More than one occurrence of a particular fncodatom within a single fncode is identical to a single occurrence of that fncodatom. Erroneous conditions are produced, with ecode="M2", when a fncodp is present with any of the sign suppression or sign placement fncodatoms ("+−" or fncodt).

$FN[UMBER] ( numexpr , fncodexpr , intexpr )

This form is identical to the two-argument form of $FNUMBER, except that numexpr is rounded to intexpr fraction digits, including possible trailing zeros, before processing any fncodatoms. If intexpr is zero, the evaluated numexpr contains no decimal point. Note: if (−1 < numexpr < 1), the result of this form of $FNUMBER has a leading zero ("0") to the left of the decimal point. Negative values of intexpr are reserved for future extensions of the $FNUMBER function.

**7.1.5.7 $GET**

$G[ET] ( glvn )

This form returns the value of the specified glvn depending on its state, defined by $DATA(glvn). The following cases are defined:

a) $D(glvn)#10 = 1

The value returned is the value of the variable specified by glvn.

b) Otherwise, the value returned is the empty string.

$G[ET] ( glvn , expr )

This form returns the value of the specified glvn depending on its state, defined by $DATA(glvn). The following cases are defined:

a) $D(glvn)#10 = 1

The value returned is the value of the variable specified by glvn.

b) Otherwise, the value returned is the value of expr.

Both glvn and expr will be evaluated before the function returns a value, so that the behavior of this function with respect to the naked indicator is well defined.

**7.1.5.8 $JUSTIFY**

$J[USTIFY] ( expr , intexpr )

This form returns the value of expr right-justified in a field of intexpr spaces. Let *m* be $LENGTH(expr) and *n* be the value of intexpr. The following cases are defined:

a) *m* '< *n*. Then the value returned is expr.

b) Otherwise, the value returned is *S*(*n*−*m*) concatenated with expr1, where *S*(*x*) is a string of *x* spaces.

$J[USTIFY] ( numexpr , intexpr1 , intexpr2 )

This form returns an edited form of the number numexpr. Let *r* be the value of numexpr after rounding to intexpr2 fraction digits, including possible trailing zeros. (If intexpr2 is the value 0, *r* contains no decimal point.) The value returned is $JUSTIFY(*r*, intexpr1). Note that if −1 < numexpr < 1, the result of $JUSTIFY does have a zero to the left of the decimal point. Negative values of intexpr2 are reserved for future extensions of the $JUSTIFY function.

**7.1.5.9 $LENGTH**

$L[ENGTH] ( expr )

This form returns an integer which is the number of characters in the value of expr. If the value of expr is the empty string, $LENGTH(expr) returns the value 0.

$L[ENGTH] ( expr1 , expr2 )

This form returns the number plus one of nonoverlapping occurrences of expr2 in expr1. If the value of expr2 is the empty string, then $LENGTH returns the value 0.

**7.1.5.10 $NAME**

$NA[ME] ( glvn )

This form returns a string value which is the namevalue denoting the named glvn. Note that naked references are permitted in the argument, but that the returned value is always a non-naked reference. If glvn includes an environment, then the namevalue shall include that environment; otherwise the namevalue shall not include an environment.

$NA[ME] ( glvn , intexpr )

This form returns a string value which is a namevalue denoting either all or part of the supplied glvn, depending on the value of intexpr. Let $NAME(glvn) applied to the supplied glvn be of the form Name(s1, s2, ..., s*n* ), considering *n* to be zero if the glvn has no subscripts, and let *m* be the value of intexpr. Then $NAME(glvn, intexpr) is defined as follows:

1) It is erroneous for *m* to be less than zero (ecode="M39").

2) If *m* = 0, the result is Name.

3) If *n* > *m*, the function returns the string returned by $NA(Name(s1, s2, ..., s*m* )).

4) Otherwise, the function returns the string returned by $NA(glvn).

**7.1.5.11 $ORDER**

$O[RDER] ( glvn )

This form returns a value which is a subscript according to a subscript ordering sequence. This ordering sequence is specified below with the aid of a function, CO, which is used for definitional purposes only, to establish the collating sequence.

CO(*s*,*t*) is defined, for strings *s* and *t*, as follows:

When *t* follows *s* in the ordering sequence or if *s* is the empty string, CO(*s*,*t*) returns *t*.

Otherwise, CO(*s*,*t*) returns *s*.

The ordering sequence is defined using the *collation algorithm* determined as follows:

a) If $ORDER refers to an ssvn, then the algorithm is determined by the value of ^$SYSTEM($SYSTEM,"COLLATE"); if that node does not exist, then the value of $GET(^$CHARACTER(^$SYSTEM,"CHARACTER"),"COLLATE")) is used.

b) If $ORDER refers to a gvn with name *global* then the algorithm is determined by the value of ^$GLOBAL("*global*","COLLATE"); if that node does not exist, then the value of $GET(^$CHARACTER(^$GLOBAL("*global"*,"CHARACTER"),"COLLATE")) is used.

c) If $ORDER does not refer to either of the above, then the algorithm is determined by the value of $GET(^$CHARACTER(^$JOB($JOB,"CHARACTER"),"COLLATE")).

d) If the resulting algorithm is the empty string, then the *collation algorithm* of the charset M (defined in Annex A) is used.

The collation value *order* of a string *subscript* using a collation algorithm *collate* may be determined by executing the expression ("S *order*="\_*collate*\_"(*subscript*)"). Two collation values are compared on a character-by-character basis using the $ASCII values (i.e. equivalent to the follows (]) operator).

Only subscripted forms of glvn are permitted. Let glvn be of the form NAME(*s1*, *s2*, ..., *sn*) where *sn* may be the empty string. Let *A* be the set of subscripts such that, *s* is in *A* if and only if:

a) CO(*sn*,*s*) = *s* and

b) $D(NAME(*s1*, *s2*, ..., *sn−1*, *s*)) is not zero.

Then $ORDER(NAME(*s1*, *s2*, ..., *sn*)) returns that value *t* in *A* such that CO(*t*,*s*) = *s* for all *s* not equal to *t*; that is, all other subscripts in *A* which follow *sn* also follow *t*.

If no such *t* exists, $ORDER returns the empty string.

$O[RDER] ( glvn , expr )

Let S be the value of expr. Then $ORDER(glvn,expr) returns:

a) If S = 1, the function returns a result identical to that returned by $ORDER(glvn).

b) If S = -1, the function returns a value which is a subscript, according to a subscript ordering sequence. This ordering sequence is specified below with the aid of a functions CO and CP, which are used for definitional purposes only, to establish the collating sequence.

CO(*s*,*t* ) is defined, for strings *s* and *t*, according to the *collation algorithm* of the specific charset.

CP(*s*,*t* ) is defined, for strings *s* and *t*, as follows:

When *t* follows *s* in the ordering sequence and *s* is not the empty string, CP(*s*,*t* ) returns *s*.

Otherwise, CP(*s*,*t* ) returns *t*.

The following cases define the ordering sequence for CP:

1) CP("",*t*) = *t*.

2) CP(*s*,*t*) = *t* if CO(*s*,*t*) = *s*; otherwise, CP(*s*,*t*) = *s*.

Only subscripted forms of glvn are permitted. Let glvn be of the form NAME(*s*1, *s*2, ..., *sn* ) where *sn* may be the empty string. Let A be the set of subscripts such that, *s* is in *A* if and only if:

1) CP(*sn*, *s*) = *s* and

2) $D(NAME(*s*1, *s*2, ..., *sn*-1, *s*)) is not zero.

Then $ORDER(NAME(*s*1, *s*2, ..., *sn* ), -1) returns that value *t* in *A* such that CP(*t*,*s*) = *t* for all *s* not equal to *t*; that is, all other subscripts in *A* which precede *s* also precede *t*.

If no such *t* exists, $ORDER(NAME(*s*1, *s*2, ..., *sn* ), -1) returns the empty string.

c) Values of S other than 1 and -1 are reserved for future extensions of the $ORDER function.

**7.1.5.12 $PIECE**

$P[IECE] ( expr1 , expr2 )

This form is defined here with the aid of a function, NF, which is used for definitional purposes only, called *find the position number following the* m*th occurrence*.

NF(*s*,*d*,*m*) is defined, for strings *s*, *d*, and integer *m*, as follows:

When *d* is the empty string, the result is zero.

When *m* '> 0, the result is zero.

When *d* is not a substring of *s*, i.e., when $F(*s*,*d*) = 0, then the result is $L(*s*) + $L(*d*) + 1.

Otherwise, NF(*s*,*d*,1) = $F(*s*,*d*).

For *m* > 1, NF(*s*,*d*,*m*) = NF($E(*s*,$F(*s*,*d*),$L(*s*)),*d*,*m*−1) + $F(*s*,*d*) − 1.

That is, NF extends $FIND to give the position number of the character to the right of the *m*th occurrence of the string *d* in *s*.

Let *s* be the value of expr1, and let *d* be the value of expr2. $PIECE(*s*,*d*) returns the substring of *s* bounded on the right but not including the first (leftmost) occurrence of *d*.

$P(*s*,*d*) = $E(*s*,0,NF(*s*,*d*,1) − $L(*d*) − 1).

$P[IECE] ( expr1 , expr2 , intexpr )

Let *m* be the integer value of intexpr. $PIECE(*s*,*d*,*m*) returns the substring of *s* bounded by but not including the *m−1*th and the *m*th occurrence of *d*.

$P(*s*,*d*,*m*) = $E(*s*,NF(*s*,*d*,*m*−1),NF(*s*,*d*,*m*) − $L(*d*) − 1).

$P[IECE] ( expr1 , expr2 , intexpr1 , intexpr2 )

Let *n* be the integer value of intexpr2. $PIECE(*s*,*d*,*m*,*n*) returns the substring of *s* bounded on the left but not including the *m−1*th occurrence of *d* in *s*, and bounded on the right but not including the *n*th occurrence of *d* in *s*.

$P(*s*,*d*,*m*,*n*) = $E(*s*,NF(*s*,*d*,*m*−1),NF(*s*,*d*,*n*) − $L(*d*) −1).

Note that $P(*s*,*d*,*m*,*m*) = $P(*s*,*d*,*m*), and that $P(*s*,*d*,1) = $P(*s*,*d*).

**7.1.5.13 $QLENGTH**

$QL[ENGTH] ( namevalue )

See 7.1.4.12 for the definition of namevalue.

This form returns a value which is derived from namevalue. If namevalue has the form NAME(*s*1, *s*2, ..., *sn*), considering *n* to be zero if there are no subscripts, then the function returns *n*.

Note that the namevalue is not "executed", and will not affect the naked indicator, nor generate an error if the namevalue represents an undefined glvn. The naked indicator will only be affected by the last gvn reference (if any) executed while evaluating the argument.

**7.1.5.14 $QSUBSCRIPT**

$QS[UBSCRIPT] ( namevalue , intexpr )

This form returns a value which is derived from namevalue. If namevalue has the form NAME( *s*1 , *s*2 , ... , *sn* ), considering *n* to be zero if there are no subscripts, and *m* is the value of intexpr, then $QSUBSCRIPT(namevalue, intexpr) is defined as follows:

a) Values of *m* less than -1 are reserved for possible future use by extension of the standard.

b) If *m* = -1, the result is the environment if namevalue includes an environment; otherwise the empty string.

c) If *m* = 0, the result is NAME without an environment even if one is present.

d) If *m* > *n*, the result is the empty string.

e) Otherwise, the result is the subscript value denoted by *sm*.

Note that the namevalue is not "executed", and will not affect the naked indicator, nor generate an error if the namevalue represents an undefined glvn. The arguments are evaluated in left to right order, and the naked indicator will only be affected by the last gvn reference (if any) executed while evaluating them.

**7.1.5.15 $QUERY**

$Q[UERY] ( glvn )

Follow these steps:

a) Let glvn be a variable reference of the form Name(*s*1, *s*2, ..., *sq* ) where *sq* may be the empty string. If glvn is unsubscripted, initialize *V* to the form Name(""); otherwise, initialize *V* to glvn.

b) If the last subscript of *V* is empty, Goto step e.

c) If $D(*V*) \ 10 = 1, append the subscript "" to *V*, i.e., *V* is Name(*s1*, *s2*, ..., *sq*, "").

d) If *V* has no subscripts, return "".

e) Let *s* = $O(*V*).

f) If *s* = "", truncate the last subscript off *V*, Goto step d.

g) If *s* '= "", replace the last subscript in *V* with *s*.

h) If $D(*V*) # 2 = 1, return *V* formatted as a namevalue.

i) Goto step c.

$Q[UERY] ( glvn , expr )

Let S be the value of expr. Then $QUERY(glvn,expr) returns:

1) If S = 1, the function returns a result identical to that returned by $QUERY(glvn).

2) If S=-1, the function returns a value which is either the empty string ("") or a namevalue according to the following steps:

a) Let glvn be a variable reference of the form Name(*s1,s2,...,sq*) where sq may be the empty string. If glvn is unsubscripted, initialize *V* to the form Name(""); otherwise, initialize *V* to glvn.

b) If the last subscript of *V* is empty, go to step e.

c) If $D(*V*)\10=1, append the subscript "" to *V*, i.e. *V* is Name(*s1,s2,...,sq,"*").

d) If *V* has no subscripts, return "".

e) Let *s* = $O(*V*,-1).

f) If *s* = "", truncate the last subscript of *V*, and go to step d.

g) If *s* '= "", replace the last subscript of *V* with *s*.

h) If $D(*V*)#2 = 1, return *V* formatted as namevalue.

I) Go to step c.

3) Values of S other than 1 or -1 are reserved for future extension of the $QUERY function.

If the value of $QUERY(glvn[,expr]) is not the empty string, and glvn includes an environment, then the namevalue shall include the environment; otherwise the namevalue shall not include an environment.

If the argument of $QUERY is a gvn, the naked indicator will become undefined and the value of $REFERENCE will become equal to the empty string.

**7.1.5.16 $RANDOM**

$R[ANDOM] ( intexpr )

This form returns a random or pseudo-random integer uniformly distributed in the closed interval

[0, intexpr−1]. If the value of intexpr is less than 1, an error condition occurs with ecode="M3".

**7.1.5.17 $REVERSE**

$RE[VERSE] ( expr )

See Clause 7 for the definition of expr.

This form returns a string whose characters are reversed in order compared to expr.

$REVERSE(EXPR) is computationally equivalent to $$REV(EXPR) which is defined by the following code

REV(E) Q $S(E="":"",1:$$REV($E(E,2,$L(E)))\_$E(E,1))

**7.1.5.18 $SELECT**

╷ ╷

$S[ELECT] ( L │ tvexpr : expr │ )

╵ ╵

This form returns the value of the leftmost expr whose corresponding tvexpr is true. The process of evaluation consists of evaluating the tvexprs, one at a time in left-to-right order, until the first one is found whose value is true. The expr corresponding to this tvexpr (and no other) is evaluated and this value is made the value of $SELECT. An error condition occurs, with ecode="M4", if all tvexprs are false. Since only one expr is evaluated at any invocation of $SELECT, that is the only expr which must have a defined value.

**7.1.5.19 $STACK**

$ST[ACK] ( intexpr )

This form returns a string as follows:

a) If intexpr is -1, returns the largest value of intexpr for which the $STACK function will return a non-empty value. Note: if $ECODE is empty then $STACK(-1)=$STACK.

b) If intexpr is 0 (zero), returns an implementation specific value indicating how this process was started.

c) If intexpr is greater than 0 (zero) and less than or equal to $STACK(-1) indicates how this level of the PROCESS-STACK was created:

1) If due to a statement, the statementword fully spelled out and in uppercase.

2) if due to an exfunc or exvar, the string "$$".

3) if due to an error, the ecode representing the error that created the result returned by $STACK(intexpr).

d) If intexpr is greater than $STACK(-1), returns an empty string.

Values of intexpr less than -1 are reserved for future extensions of the $STACK function.

$ST[ACK] ( intexpr , stackcodexpr )

stackcodexpr ::= expr V stackcode

╷ ╷

│ PLACE │

stackcode ::= │ MCODE │

│ ECODE │

This form returns information about the action that created this level of the PROCESS-STACK as follows:

stackcode Returned String

ECODE the list of any ecodes added at this level.

MCODE the value (in the case of an XECUTE) or the line for the location identified by $STACK(intexpr,"PLACE"). If the line is not available, an empty string is returned.

PLACE the location of a statement at the intexpr level of the PROCESS-STACK as follows:

a) if intexpr is not equal to $STACK and $STACK(intexpr,"ECODE") would return the empty string, the last statement executed.

b) if intexpr is equal to $STACK and $STACK(intexpr,"ECODE") would return the empty string, the currently executing statement.

c) if $STACK(intexpr,"ECODE") would return a non-empty string, the last statement to start execution while $STACK(intexpr,"ECODE") would have returned the empty string.

The location is in the form:

place SP + eoffset

place ::= │ [ label ] [ + offset ] [ ^ VB environment VB routinename ] │

│ @ │

eoffset ::= intlit

In place, the first case is used to identify the line being executed at the time of creation of this level of the PROCESS-STACK. The second case (@) shows the point of execution occurring in an XECUTE.

eoffset is an offset into the code or data identified by place at which the error occurred. The value might point to the first or last character of a "token" just before or just after a "token", or even to the statement or block in which the error occurred. Implementors should provide as accurate a value for eoffset as practical.

All values of stackcode beginning with the letter Z are reserved for the implementation. All other values of stackcode are reserved for future extensions of the $STACK function. stackcodes differing only in the use of corresponding upper and lower case letters are equivalent.

**7.1.5.20 $TEXT**

$T[EXT] ( textarg )

│ + intexpr [ ^ routineref ] │

│ │

textarg ::= │ textref │

│ │

│ @ expratom V textarg │

This form returns a string whose value is the contents of the line specified by the argument. Specifically, the entire line, with eol deleted, is returned.

If the argument of $TEXT is an textref, the line denoted by the textref is specified. If textref does not contain dlabel then the line denoted is the first line of the routine. If the argument is of the form + intexpr [ ^ routineref ], two cases are defined. If the value of intexpr is greater than 0, the intexprth line of the routine is specified; if the value of intexpr is equal to 0, the routinename of the routine is specified. An error condition occurs, with ecode="M5", if the value of intexpr is less than 0. In all cases, if no routine is explicitly specified, the currently-executing routine is used.

If no such line as that specified by the argument exists, an empty string is returned. If the line specification is ambiguous, the results are not defined.

If a Character Set Profile input-transform is in effect, then the string is modified in accordance with the transform.

**7.1.5.21 $TRANSLATE**

$TR[ANSLATE] ( expr1 , expr2 )

Let *s* be the value of expr1, $TRANSLATE(expr1,expr2) returns an edited form of *s* in which all characters in *s* which are found in expr2 are removed.

$TR[ANSLATE] ( expr1 , expr2 , expr3 )

Let *s* be the value of expr1, $TRANSLATE(expr1,expr2,expr3) returns an edited form of *s* in which all characters in *s* which are found in expr2 are replaced by the positionally corresponding character in expr3. If a character in *s* appears more than once in expr2 the first (leftmost) occurrence is used to positionally locate the translation.

Translation is performed once for each character in *s*. Characters which are in *s* that are not in expr2 remain unchanged. Characters in expr2 which have no corresponding character in expr3 are deleted from *s* (this is the case when expr3 is shorter than expr2).

Note: If the value of expr2 is the empty string, no translation is performed and *s* is returned unchanged.

**7.1.5.22 $VIEW**

$V[IEW] ( unspecified )

makes available to the implementor a call for examining machine-dependent information. It is to be understood that routines containing occurrences of $VIEW may not be portable.

**7.1.5.23 $Z**

$Z[unspecified] ( unspecified )

is the initial letter reserved for defining non-standard intrinsic functions. This requirement permits the unused function names to be reserved for future use.

7.1.6 Omega Standard Library

7.1.6.1 Library definitions

A library consists of a set of libraryelements - functions and data which are accessed from Omega and which have unique names within the library. The access method for each libraryelement is the external calling syntax, which normally has no side-effects.

A library is defined as being either mandatory or optional. library names starting with a Z are reserved for implementors. library names starting with a Y are reserved for users. All other unused library names are reserved for future use.

The Omega Standard Library is the set of library definitions in this standard.

The following librarys are defined:

**7.1.6.1.1 Mandatory Libraries**

CHARACTER

MATH

STRING

**7.1.6.1.2 Optional Libraries**

7.1.6.2 Library Element Definitions

The definition of a libraryelement states which library the element belongs to, return value type, and full specification.

libraryelement names starting with a Z are reserved for implementors. libraryelement names starting with a Y are reserved for users. All other unused libraryelement names are reserved for future use.

A libraryelement definition is of the form:

libraryelementdef ::= libraryelement ^ library libraryresult

[ ( L libraryparam ) ]

libraryparam ::= [ . ] name [ : [ libdatatype ] [ : libraryopt ] ]

libraryresult ::= [ : libdatatype ]

╷ BOOLEAN ╷

│ COMPLEX │

│ INTEGER │

libdatatype ::= │ MATRIX │

│ NAME │

│ REAL │

│ STRING │

│ Z[unspecified]│

libraryopt ::= │ M │

│ O │

If a libraryparam starts with a period then this parameter is passed by reference.

Z is the initial letter reserved for implementation specific libdatatypes. All other values for libdatatypes are reserved for future expansion of the standard.

Input and output values to libraryelements undergo the appropriate data interpretation below:

a) For BOOLEAN see 7.1.4.7 (Truth-value interpretation).

b) COMPLEX numbers are represented as strings of the format REAL\_"%"\_REAL (that is, two REAL numbers separated by the % character). Any string has a value when interpreted as a complex number. The real part of the complex number is the numeric interpretation of the first "%" piece and the imaginary part is the numeric interpretation of the second "%" piece. The canonic representation of a complex number is a string created by concatenating the canonic numerical representation of the real part, a percent sign, and the canonic numerical representation of the imaginary part.

c) For INTEGER see 7.1.4.6 (Integer interpretation).

d) MATRIX values are represented as sparse arrays, in which only the defined nodes with two integer-valued subscripts will be accessed or modified. Any other nodes in the arrays are not considered part of the matrix, and do not affect and are not affected by the matrix functions. The string *A[R,C]* denotes a matrix *A* having *R* rows and *C* columns. The canonic representation of a matrix value contains no non-matrix array nodes.

e) For NAME see 7.1.4.12 (Name value namevalue).

f) For REAL see 7.1.4.5 (Numeric interpretation of data).

g) STRING is a string made up of any characters and not constrained in format.

If no libdatatype is specified for a libraryparam or libraryresult then the libdatatype defaults to STRING.

If no libraryopt is specified then the libraryparam is M (mandatory). A libraryopt of O specifies that the libraryparam is optional.

Unless otherwise specified in their definitions, library elements are assumed to have the standard domain and range for their function, and no side effects. For each library element with a standard domain, *all* of its libraryparams can assume any valid values of their respective libdatatypes. Similarly, for each with a standard range, the libraryresult and *all* of its output libraryparams can assume any valid values of their respective libdatatypes.

7.1.6.3 Availability of library elements

An implementation of Omega shall

a) provide the mandatory librarys defined in this standard

and

b) provide a means by which replacement definitions in routines of libraryelements can be installed so that a routine can access them as if they were part of the implementation. An implementation may additionally provide a means by which non-M code can be installed to implement libraryelements.

An implementation may also provide a means by which specific librarys or libraryelements of the Omega Standard Library are only optionally installed.

7.1.6.4 CHARACTER Library elements

7.1.6.4.1 $%COLLATE^CHARACTER

COLLATE^CHARACTER : STRING ( A : STRING , CHARMOD : : O )

$%COLLATE^CHARACTER returns the collation value of a string according to the specification of the collation algorithm. CHARMOD is either a Character Set Profile specification in the form charset or a global name specification in the form ^name. If CHARMOD is a Character Set Profile then the collation algorithm used is that specified in ^$CHARACTER for the profile. If CHARMOD is a global name then the collation algorithm used is that specified in ^$GLOBAL for that name. If CHARMOD is not specified, or the node specified above does not exist, then the collation algorithm used is the default process collating algorithm.

7.1.6.4.2 $%COMPARE^CHARACTER

COMPARE^CHARACTER : INTEGER ( A : STRING , B : STRING , CHARMOD : : O )

$%COMPARE^CHARACTER compares two strings according to the specification of the collation algorithm, and returns:

-1 if A compares before B

0 if A compares the same as B

1 if A compares after B

CHARMOD is either a Character Set Profile specification in the form charset or a global name specification in the form ^name. If CHARMOD is a Character Set Profile then the two strings are compared using the collation algorithm specified in ^$CHARACTER for the profile. If CHARMOD is a global name then the two strings are compared using the collation algorithm specified in ^$GLOBAL for that name. If CHARMOD is not specified, or the node specified above does not exist, then the two strings are compared using the default process collation algorithm.

7.1.6.5 MATH Library elements

7.1.6.5.1 $%ABS^MATH

ABS^MATH : REAL ( X : REAL )

$%ABS^MATH returns the absolute value of its parameter.

7.1.6.5.2 $%ARCCOS^MATH

ARCCOS^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCOS^MATH returns the value of the trigonometric arccosine in radians of X; 0 ≤ $%ARCCOS^MATH(X) ≤ ∏. The number of significant digits in the arccosine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. When X < -1 or X > 1, an error condition occurs with ecode = "M28".

7.1.6.5.3 $%ARCCOSH^MATH

ARCCOSH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCCOSH^MATH returns the value of the hyperbolic arccosine in radians of X; $%ARCCOSH^MATH(X) ≥ 0. The number of significant digits in the hyperbolic arccosine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. When X < 1, an error condition occurs with ecode = "M28".

7.1.6.5.4 $%ARCCOT^MATH

ARCCOT^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCCOT^MATH returns the value of the trigonometric arccotangent in radians of X; 0 < $%ARCCOS^MATH(X) < ∏. The number of significant digits in the arccotangent is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.5 $%ARCCOTH^MATH

ARCCOTH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCCOTH^MATH returns the value of the hyperbolic arccotangent in radians of X; $%ARCCOTH^MATH(X) < 0 when X ≤ 11, and $%ARCCOTH^MATH(X) >0 when X ≥ 1. The number of significant digits in the hyperbolic arccotangent is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. When -1 ≤ X ≥ 1, an error condition occurs with ecode = "M28".

7.1.6.5.6 $%ARCCSC^MATH

ARCCSC^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCCSC^MATH returns the value of the trigonometric arccosecant in radians of X; 0 ≤ $%ARCCSC^MATH(X) ≤ ∏. The number of significant digits in the arccosecant is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. When X < -1 or X > 1, an error condition occurs with ecode = "M28".

7.1.6.5.7 $%ARCSEC^MATH

ARCSEC^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCSEC^MATH returns the value of the trigonometric arcsecant in radians of X; 0 ≤ $%ARCSEC^MATH(X) ≤ ∏. The number of significant digits in the arcsecant is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. When X < -1 or X > 1, an error condition occurs with ecode = "M28".

7.1.6.5.8 $%ARCSIN^MATH

ARCSIN^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCSIN^MATH returns the value of the trigonometric arcsine in radians of X; -∏/2 ≤ $%ARCSIN^MATH(X) ≤ ∏/2. The number of significant digits in the arcsine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. When X < -1 or X > 1, an error condition occurs with ecode = "M28".

7.1.6.5.9 $%ARCSINH^MATH

ARCSINH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCSINH^MATH returns the value of the hyperbolic arcsine in radians of X. The number of significant digits in the hyperbolic arcsine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.10 $%ARCTAN^MATH

ARCTAN^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCTAN^MATH returns the value of the trigonometric arctangent in radians of X; |$%ARCTAN^MATH(X)| ≤ ∏/2, 0 ≤ $%ARCTAN^MATH(X) ≤ ∏ when X ≥ 0, and -∏ ≤ $%ARCTAN^MATH(X) ≤ 0 when X ≤ 0. The number of significant digits in the arctangent is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.11 $%ARCTANH^MATH

ARCTANH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%ARCTANH^MATH returns the value of the hyperbolic artangent in radians of X. The number of significant digits in the hyperbolic artangent is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. When X ≤ -1 or X ≥ 1, an error condition occurs with ecode = "M28".

7.1.6.5.12 $%CABS^MATH

CABS^MATH : REAL ( Z : REAL )

$%CABS^MATH returns the absolute value of the complex number Z.

7.1.6.5.13 $%CADD^MATH

CADD^MATH : COMPLEX ( X : COMPLEX , Y : COMPLEX )

$%CADD^MATH returns the sum of X + Y, where X and Y are complex numbers.

7.1.6.5.14 $%CCOS^MATH

CCOS^MATH : COMPLEX ( Z : COMPLEX , PREC : INTEGER : O )

$%CCOS^MATH returns the value of the trigonometric cosine cos(Z) of the angle Z in radians, -1 ≤ $%CCOS^MATH(Z) ≤ 1. Z is interpreted as a complex number. The number of significant digits in the complex cosine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.15 $%CDIV^MATH

CDIV^MATH : COMPLEX ( X : COMPLEX , Y : COMPLEX )

$%CDIV^MATH returns the value X / Y, where X and Y are complex numbers. If the complex numeric interpretation of Y is equal to "0%0", an error condition occurs with ecode = "M9"

7.1.6.5.16 $%CEXP^MATH

CEXP^MATH : COMPLEX ( Z : COMPLEX , PREC : INTEGER : O )

$%CEXP^MATH returns the value of *e* raised to the power of the complex number Z. The number of significant digits in the complex exponent is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.17 $%CLOG^MATH

CLOG^MATH : COMPLEX ( Z : COMPLEX , PREC : INTEGER : O )

$%CLOG^MATH returns the logarithm of the complex number Z; Re $%CLOG^MATH(Z) can be any number, -∏ ≤ Im $%CLOG^MATH(Z) ≤ ∏. The number of significant digits in the complex logarithm is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. If Im Z = 0, then Re Z > 0.

7.1.6.5.18 $%CMUL^MATH

CMUL^MATH : COMPLEX ( X : COMPLEX , Y : COMPLEX )

$%CMUL^MATH returns the value of X \* Y, where X and Y are complex numbers.

7.1.6.5.19 $%COMPLEX^MATH

COMPLEX^MATH : COMPLEX ( X : REAL )

$%COMPLEX^MATH returns the complex representation of the number specified in X.

7.1.6.5.20 $%CONJUG^MATH

CONJUG^MATH : COMPLEX ( Z : COMPLEX )

$%CONJUG^MATH returns the value of the conjugate of the complex number Z.

7.1.6.5.21 $%COS^MATH

COS^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%COS^MATH returns the value of the trigonometric cosine in radians of X; -1 ≤ $%COS^MATH(X) ≤ 1. The number of significant digits in the cosine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.22 $%COSH^MATH

COSH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%COSH^MATH returns the value of the hyperbolic cosine in radians of X; $%COSH^MATH(X) ≥ 1. The number of significant digits in the hyperbolic cosine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.23 $%COT^MATH

COT^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%COT^MATH returns the value of the trigonometric cotangent in radians of X. The number of significant digits in the cotangent is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.24 $%COTH^MATH

COTH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

%COTH^MATH returns the value of the hyperbolic cotangent in radians of X; $%COTH^MATH(X) < -1 when X < 0 and $%COTH^MATH(X) > 1 when X > 0. The number of significant digits in the hyperbolic cotangent is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.25 $%CPOWER^MATH

CPOWER^MATH : COMPLEX ( Z : COMPLEX , X : COMPLEX , PREC : INTEGER : O )

$%CPOWER^MATH returns the value of Z raised to the power of X, where Z and X are complex numbers. The number of significant digits in the complex power is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.26 $%CSC^MATH

CSC^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%CSC^MATH returns the value of the trigonometric cosecant in radians of X; $%CSC^MATH(X) ≤ -1 or $%CSC^MATH(X) ≥ 1. The number of significant digits in the cosecant is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.27 $%CSCH^MATH

CSCH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%CSCH^MATH returns the value of the hyperbolic cosecant in radians of X. The number of significant digits in the hyperbolic cosecant is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.28 $%CSIN^MATH

CSIN^MATH : COMPLEX ( Z : COMPLEX , PREC : INTEGER : O )

$%CSIN^MATH returns the value of the trigonometric sine sin(Z) of the angle Z in radians, where Z is interpreted as a complex number; -1 ≤ Re $%CSIN^MATH(Z) ≤ 1, -1 ≤ Im $%CSIN^MATH(Z) ≤ 1. The number of significant digits in the complex sine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.29 $%CSUB^MATH

CSUB^MATH : COMPLEX ( X : COMPLEX , Y : COMPLEX )

$%CSUB^MATH returns the value of X - Y, where X and Y are complex numbers.

7.1.6.5.30 $%DECDMS^MATH

DECDMS^MATH : STRING ( X : REAL , PREC : INTEGER : O )

$%DECDMS^MATH returns a string, containing the ̊ ʹ ʺ notation for the angle that is specified in X in degrees. Since the symbols for degrees, minutes, and seconds are not in the ASCII set, the fields in the result-value are separated by colons (":"); the value of the first part is an integer in the range [0,359]; the value in the second part is an integer in the range [0,59]; the value in the third part is a real number in the range [0,60). The optional parameter PREC specifies the precision to which X is rounded before the conversion takes place. If not specified, a default value of 5 digits is assumed for PREC.

7.1.6.5.31 $%DEGRAD^MATH

DEGRAD^MATH : REAL ( X : REAL )

$%DEGRAD^MATH returns the value in radians that is equal to the angle specified in X in degrees. A full circle is 2∏ radians, or 360 degrees.

7.1.6.5.32 $%DMSDEC^MATH

DMSDEC^MATH : REAL ( X : STRING )

$%DMSDEC^MATH returns the value in degrees that is equal to the angle specified in X in ̊ ʹ ʺ notation. Since the symbols for degrees, minutes, and seconds are not in the ASCII set, the three fields in X must be separated by colons (":"); the value of the first part is an integer in the range [0,359]; the value in the second part is an integer in the range [0,59]; the value in the third part is a real number in the range [0,60). Any further ":" separated parts in the value of X are ignored.

7.1.6.5.33 $%E^MATH

E^MATH : REAL ( )

$%E^MATH returns the value of Euler's number, approximated to at least 15 significant digits.

7.1.6.5.34 $%EXP^MATH

EXP^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%EXP^MATH returns the value of *e* to the power of X. The exponentiation is approximated with as many significant digits as specified by the optional parameter PREC. If not specified, a default value of 11 is assumed for PREC.

7.1.6.5.35 $%LOG^MATH

LOG^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%LOG^MATH returns the Naperian logarithm of X. The number of significant digits in the logarithm is specified by the optional parameter PREC. If not supplied, a default value of 11 digits is assumed for PREC. When X is less than or equal to 0, an error condition occurs with ecode = "M28".

7.1.6.5.36 $%LOG10^MATH

LOG10^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%LOG10^MATH returns the Briggsian logarithm of X. The number of significant digits in the logarithm is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. When X is less than or equal to 0, an error condition occurs with ecode = "M28".

7.1.6.5.37 $%MTXADD^MATH

MTXADD^MATH : BOOLEAN ( .A : MATRIX , .B : MATRIX , .R : MATRIX , ROWS : INTEGER , COLS : INTEGER )

$%MTXADD^MATH adds matrix B[ROWS,COLS] to matrix A[ROWS,COLS], and stores the result into matrix R[ROWS,COLS]. It is permissible that the actual parameter for matrix R is equal to either of the actual parameters for matrices A and B. The return value is 1 if both matrices A and B exist; or 0 if a) there are no defined values in one or both of the matrices A and B, or b) when ROWS < 1 or COLS < 1.

7.1.6.5.38 $%MTXCOF^MATH

MTXCOF^MATH : REAL ( .A : MATRIX , I : INTEGER , K : INTEGER , N : INTEGER )

$%MTXCOF^MATH computes the cofactor in matrix A[N,N] for element A(I,K). The return value is the value of the cofactor.

7.1.6.5.39 $%MTXCOPY^MATH

MTXCOPY^MATH : BOOLEAN ( .A: MATRIX , .R: MATRIX , ROWS : INTEGER , COLS : INTEGER )

$%MTXCOPY^MATH copies the matrix A[ROWS,COLS] into the matrix R[ROWS,COLS]. The return value is 1 if matrix A exists; or 0 if a) there are no defined values in the matrix A, or b) when ROWS < 1 or COLS < 1.

7.1.6.5.40 $%MTXDET^MATH

MTXDET^MATH : REAL ( .A : MATRIX , N : INTEGER )

$%MTXDET^MATH computes the determinant of matrix A[N,N]. The return value is the value of the determinant; or “” (empty) if a) the determinant cannot be computed, or b) when N < 1.

7.1.6.5.41 $%MTXEQU^MATH

MTXEQU^MATH : BOOLEAN ( .A : MATRIX , .B : MATRIX , .R : MATRIX , N : INTEGER , M : INTEGER )

$%MTXEQU^MATH solves the matrix-equation A[M,N] \* R[M,N] = B[M,N], with matrix R[M,N] being the unknown to be resolved. The return value is 1 if a solution to the equation can be computed, 0 if it cannot, or “” (the empty string) if M < 1 or N < 1.

7.1.6.5.42 $%MTXINV^MATH

MTXINV^MATH : BOOLEAN ( .A : MATRIX , .R : MATRIX , N : INTEGER )

$%MTXINV^MATH inverts matrix A[N,N] into matrix R[N,N]. It is permissible that the actual parameter for matrix R is equal to the actual parameter for matrix A. The return value is 1 if matrix

A has been inverted into matrix R; or 0 if a) no inverse matrix can be computed, or b) when N <= 0.

7.1.6.5.43 $%MTXMUL^MATH

MTXMUL^MATH : BOOLEAN ( .A : MATRIX , .B : MATRIX , .R : MATRIX , M : INTEGER , L : INTEGER , N : INTEGER )

$%MTXMUL^MATH multiplies matrix A[M,L] with matrix B[L,N]; the result is stored into matrix R[M,N]. The actual parameter for matrix R may not be equal to the actual parameter for matrix A, or the actual parameter for matrix B. The return value is 1 if both matrices A and B exist; or 0 if a) there are no defined values in one or both of the matrices A and B, or b) when L < 1 or M < 1 or N < 1.

7.1.6.5.44 $%MTXSCA^MATH

MTXSCA^MATH : BOOLEAN ( .A : MATRIX , .R : MATRIX , ROWS : INTEGER , COLS : INTEGER , S : REAL )

$%MTXSCA^MATH multiplies scalar value S with matrix A[ROWS,COLS], and stores the result into matrix R[ROWS,COLS]. It is permissible that the actual parameter for matrix R is equal to the actual parameter for matrix A. The return value is 1 if matrix A exists; or 0 if a) there are no defined values in the matrix A, or b) when ROWS < 1 or COLS < 1.

7.1.6.5.45 $%MTXSUB^MATH

MTXSUB^MATH : BOOLEAN ( .A : MATRIX , .B : MATRIX , .R : MATRIX , ROWS : INTEGER , COLS : INTEGER)

$%MTXSUB^MATH subtracts matrix B[ROWS,COLS] from matrix A[ROWS,COLS], and stores the result into matrix R[ROWS,COLS]. It is permissible that the actual parameter for matrix R is equal to either of the actual parameters for matrices A and B. The return value is 1 if both matrices A and B exist; or 0 if a) there are no defined values in one or both of the matrices A and B, or b) when ROWS < 1 or COLS < 1.

7.1.6.5.46 $%MTXTRP^MATH

LOG10^MATH : BOOLEAN ( .A : MATRIX , .R : MATRIX , M : INTEGER , N : INTEGER )

$%MTXTRP^MATH transposes matrix A[M,N] into matrix R[N,M]. It is permissible that the actual parameter for matrix R is equal to the actual parameter for matrix A. The return value is 1 if matrix A exists; or 0 if a) there are no defined values in the matrix A, or b) when M < 1 or N < 1.

7.1.6.5.47 $%MTXUNIT^MATH

MTXUNIT^MATH : BOOLEAN ( .R : MATRIX , N : INTEGER , SPARSE : BOOLEAN : O)

$%MTXUNIT^MATH creates matrix R[N,N] as a unit matrix. If the value of the optional parameter SPARSE is 1 (*true*), a sparse unit matrix will be created, i.e., only the diagonal elements of the result matrix will be defined. The return value is 1 if a unit matrix can be created; or 0 if a) it cannot be created, or b) when N < 1.

7.1.6.5.48 $%PI^MATH

PI^MATH : REAL ( )

$%PI^MATH returns the value of ∏ (pi), approximated to at least 15 significant digits.

7.1.6.5.49 $%RADDEG^MATH

RADDEG^MATH : REAL ( X : REAL )

%RADDEG^MATH returns the value in degrees that is equal to the angle specified in X in radians. A full ircle is 2∏ radians, or 360 degrees.

7.1.6.5.50 $%SEC^MATH

SEC^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%SEC^MATH returns the value of the trigonometric secant in radians of X; $%SEC^MATH(X) ≤ -1 or $%SEC^MATH(X) ≥ 1. The number of significant digits in the secant is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.51 $%SECH^MATH

SECH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%SECH^MATH returns the value of the hyperbolic secant in radians of X; 0 < $%SECH^MATH(X) ≤ 1. The number of significant digits in the hyperbolic secant is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.52 $%SIGN^MATH

SIGN^MATH : REAL ( X : REAL )

$%SIGN^MATH returns 0, -1, or 1, depending on the value of X. If X < 0, SIGN^MATH returns -1. If X equals 0, it returns 0. If X > 0, it returns 1.

7.1.6.5.53 $%SIN^MATH

SIN^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%SIN^MATH returns the value of the trigonometric sine in radians of X; -1 ≤ $%SIN^MATH(X) ≤ 1. The number of significant digits in the sine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.54 $%SINH^MATH

SINH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%SINH^MATH returns the value of the hyperbolic sine in radians of X. The number of significant digits in the hyperbolic sine is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.55 $%SQRT^MATH

SQRT^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%SQRT^MATH returns the square root of X. The number of significant digits in the square root is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC. When X is less than 0, an error condition occurs with ecode = "M28".

7.1.6.5.56 $%TAN^MATH

TAN^MATH : REAL ( X : REAL , PREC : INTEGER : O )

%TAN^MATH returns the value of the trigonometric tangent in radians of X. The number of significant digits in the tangent is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.5.57 $%TANH^MATH

TANH^MATH : REAL ( X : REAL , PREC : INTEGER : O )

$%TANH^MATH returns the value of the hyperbolic tangent in radians of X; -1 < $%TAN^MATH(X) ≤ 1. The number of significant digits in the hyperbolic tangent is specified by the optional parameter PREC. If not specified, a default value of 11 digits is assumed for PREC.

7.1.6.6 STRING Library Elements

7.1.6.6.1 $%PRODUCE^STRING

PRODUCE^STRING : STRING ( IN : STRING , .SPEC , MAX : INTEGER : O )

$%PRODUCE^STRING returns a copy of IN transformed by repeated prioritized substring replacement. For each element I in order by I's collation sequence, where SPEC(I,1) = FIND and SPEC(I,2) = OUT, the function will scan IN for occurrences of the substring FIND. For each FIND found in IN, whether or not any of the characters in the substring have already been replaced the matching substring in IN is replaced with OUT. Processing of each substring is complete when no more occurrences of that substring can be found in IN; when all of the substrings in SPEC have been processed in order, the scanning and replacement begins again wit the first I in SPEC; only when none of the substrings can be found is the resulting string is returned. The optional parameter MAX sets a maximum number of replacements $%PRODUCE^STRING will make before ending processing and returning the result.

7.1.6.6.2 $%REPLACE^STRING

REPLACE^STRING : STRING ( IN : STRING , .SPEC )

$%REPLACE^STRING returns a copy of IN transformed by prioritized substring replacement. For each element I in order by I's collation sequence, where SPEC(I,1) = FIND and SPEC(I,2) = OUT, the function will scan IN for occurrences of the substring FIND. For each FIND found in IN, if none of the characters in the substring have already been replaced then the matching substring in IN is replaced with OUT; otherwise, the found substring is ignored. Processing of each substring is complete when no more unmodified occurrences of that substring can be found in IN; when all of the substrings in SPEC have been processed in order, the resulting string is returned.

**7.2 Expression tail exprtail**

│ │ binaryop │ expratom │

exprtail ::= │ │ ['] truthop │ │

│ │

│ ['] ? pattern │

The order of evaluation is as follows:

a) Evaluate the left-hand expratom.

b) If an exprtail is present immediately to the right, evaluate its expratom or pattern and apply its operator.

c) Repeat step b. as necessary, moving to the right.

In the language of operator precedence, this sequence implies that all binary string, arithmetic, and truth-valued operators are at the same precedence level and are applied in left-to-right order.

Any attempt to evaluate an expratom containing an lvn, gvn, ssvn, or svn with an undefined value is erroneous. A reference to a lvn with an undefined value causes an error condition with ecode="M6". A reference to a gvn with an undefined value causes an error condition with ecode="M7". A reference to an svn with an undefined value causes an error condition with ecode="M8". A reference to an ssvn with an undefined value, where the semantics of that action are not specified for that specific ssvn, causes an error condition with ecode = "M60".

**7.2.1 Binary operator** binaryop

│ │ (Note: underscore)

│ + │

│ - │ (Note: hyphen)

binaryop ::= │ \* │

│ / │

│ # │

│ \ │

│ \*\* │

**7.2.1.1 Concatenation operator**

The underscore symbol \_ is the concatenation operator. It does not imply any numeric interpretation. The value of *A*\_*B* is the string obtained by concatenating the values of *A* and *B*, with *A* on the left.

**7.2.1.2 Arithmetic binary operators**

The binary operators + − \* / \ # \*\* are called the arithmetic binary operators. They operate on the numeric interpretations of their operands, and they produce numeric (in one case, integer) results.

+ produces the algebraic sum.

− produces the algebraic difference.

\* produces the algebraic product.

/ produces the algebraic quotient. Note that the sign of the quotient is negative if and only if one operand is positive and one operand is negative. Division by zero causes an error condition with ecode="M9".

\ produces the integer interpretation of the result of the algebraic quotient.

# produces the value of the left operand modulo the right operand. It is defined only for nonzero values of its right operand, as follows.

*A* # *B* = *A* − (*B* \* floor(*A*/*B*))

where floor (*x*) = the largest integer '> *x*.

A value of 0 (zero) for *B* will produce an error condition with ecode="M9".

\*\* produces the exponentiated value of the left operand, raised to the power of the right operand. Results producing complex numbers (eg, even numbered roots of negative numbers) are not defined.

**7.2.2 Truth operator truthop**

truthop ::= │ relation │

│ logicalop │

**7.2.2.1 Relational operator relation**

│ = │

│ < │

│ <= │

│ > │

│ >= │

relation ::= │ [ │

│ ] │

│ ]= │

│ ]] │

│ ]]=│

The operators =, <, <=, >, >=, [, ], ]=, ]], and ]]= produce the truth value 1 if the relation between their operands which they express is true, and 0 otherwise. The dual operators 'relation are defined by:

*A* 'relation *B* has the same value as '(*A* relation *B*).

**7.2.2.2 Numeric relations**

The inequalities <, <=, >, and >= operate on the numeric interpretations of their operands; they denote the conventional algebraic *less than*, *less than or equal to*, *greater than*, and *greater than or equal to*.

**7.2.2.3 String relations**

The relations =, [, ], ]=, ]], and ]]= do not imply any numeric interpretation of either of their operands.

The relation = tests string identity. If the operands are not known to be numeric and numeric equality is to be tested, the programmer may apply an appropriate unary operator to the nonnumeric operands. If both arguments are known to be in numeric form (as would be the case, for example, if they resulted from the application of any operator except \_), application of a unary operator is not necessary. The uniqueness of the numeric representation guarantees the equivalence of string and numeric equality when both operands are numeric. Note, however, that the division operator / may produce inexact results, with the usual problems attendant to inexact arithmetic.

The relation [ is called *contains*. *A* [ *B* is true if and only if *B* is a substring of *A*; that is, *A* [ *B* has the same value as ''$FIND(*A*,*B*). Note that the empty string is a substring of every string.

The relation ] is called *follows*. *A* ] *B* is true if and only if *A* follows *B* in the sequence, defined here. *A* follows *B* if and only if any of the following is true.

a) *B* is empty and *A* is not.

b) Neither *A* nor *B* is empty, and the leftmost character of *A* follows (i.e., has a numerically greater $ASCII value than) the leftmost character of *B*.

c) There exists a positive integer *n* such that *A* and *B* have identical heads of length *n*, (i.e., $EXTRACT(*A*,1,*n*) = $EXTRACT(*B*,1,*n*)) and the remainder of *A* follows the remainder of *B* (i.e., $EXTRACT(*A*,*n*+1,$LENGTH(*A*)) follows $EXTRACT(*B*,*n*+1,$LENGTH(*B*))).

The relation ]= is called *follows or equals*. *A*]=*B* is true if and only if *A* follows *B* as defined above or *A* is identical to *B*.

The relation ]] is called *sorts after*. *A*]]*B* is true if and only if *A* follows *B* in the subscript ordering sequence defined by the single argument $ORDER function as if that $ORDER refers to a lvn.

The relation ]]= is called *sorts after or equals*. *A*]]=*B* is true if and only if *A* sorts after *B* as defined above or *A* is identical to *B*.

**7.2.2.4 Logical operator logicalop**

logicalop ::= │ & │

│ ! │

│ !! │

The operators !, !!, and & are called logical operators. (They are given the names *or*, *exclusive or*, and *and*, respectively.) They operate on the truth-value interpretations of their arguments, and they produce truth-value results.

*A* ! *B* = ( 0 if both *A* and *B* have the value 0 )

( 1 otherwise )

*A* !! *B* = ( 0 if both *A* and *B* have the value 0 or if both A and B have the value 1 )

( 1 otherwise )

*A* & *B* = ( 1 if both *A* and *B* have the value 1 )

( 0 otherwise )

The dual operators '& and '! are defined by:

*A* '& *B* = '(*A* & *B*)

*A* '! *B* = '(*A* ! *B*)

*A* '!! *B* = '(*A* !! *B*)

**7.2.3 Pattern match pattern**

The pattern match operator ? tests the form of the string which is its left-hand operand. *S* ? *P* is true if and only if *S* is a member of the class of strings specified by the pattern *P*.

A pattern is a concatenated list of pattern atoms.

│ patatom ... │

pattern ::= │ │

│ @ expratom V pattern │

Assume that pattern has *n* patatoms. *S* ? pattern is true if and only if there exists a partition of *S* into *n* substrings

*S* = *S*1 *S*2 ... *Sn*

such that there is a one-to-one order-preserving correspondence between the *Si* and the pattern atoms, and each *Si* *satisfies* its respective pattern atom. Note that some of the *Si* may be empty.

Each pattern atom consists of a repeat count repcount, followed by either a pattern code patcode or a string literal strlit. A substring *Si* of *S* satisfies a pattern atom if it, in turn, can be decomposed into a number of concatenated substrings, each of which satisfies the associated patcode or strlit.

│ patcode │

│ │

patatom ::= repcount │ patstr │

│ │

│ alternation │

│ intlit │

repcount ::= │ │

│ [ intlit1 ] . [ intlit2 ] │

│ Y patnonY Y │

patcode ::= [ ' ] │ Z patnonZ Z │ ...

│ patnonYZ │

│ OB charspec CB │

patnonY ::= any of the characters in ident except Y

patnonZ ::= any of the characters in ident except Z

patnonYZ ::= any of the characters in ident except Y and Z

charspec ::= strconst1 [ : strconst2 ]

│ $C [ HAR ] ( L numlit ) │

strconst ::= │ │

│ strlit │

patstr ::= [ ' ] strlit

alternation ::= ( L patgrp )

patgrp ::= patatom ...

patcodes beginning with the initial letter Y are available for use by Omega programmers. patcodes beginning with the initial letter Z are available for use by implementors. patcodes are specified in Character Set Profiles.

a If a patcode has the form of a charspec, determination of whether a character belongs to the patcode is made as follows: A character belongs to a charspec containing only one strconst if it is contained in the string represented by that strconst. A character belongs to a charspec containing two strconsts if it is (inclusively) between them. Formally, *X* is a member of *S* if *S*[*X*, and *X* is a member of *S1*:*S2* if *S1* does not trail *X* and *X* does not trail *S2*, but the check against the value of *S2* will be omitted if *S2* is the empty string. If *S2* is present, then neither *S1* nor *S2* may contain more than one character.

If a strconst is of the form $C[HAR}( ... ), then it has the same value as the result of the function $CHAR called with the same parameters. Use of upper, lower, or mixed case in the name $CHAR is permitted

b Otherwise, patcodes differing only in the use of corresponding upper and lower case letters are equivalent. If the apostrophe is not present in a given patcode, the patcode is satisfied by any single character in the union of the classes of characters represented, each class denoted by its own patcode letter. If the apostrophe is present, the patcode is satisfied by any single character which is not in the union of the classes of characters represented. Whether or not a specific character belongs to a patcode class is determined by a process' Character Set Profile ( charset ).

An alternation is satisfied if any one of its patgrp components individually matches the corresponding Si.

Each patstr in which an apostrophe is not present is satisfied by, and only by, the value of strlit. Each patstr in which an apostrophe is present is satisfied by any string of the same length as strlit which is not identical to strlit.

If repcount has the form of an indefinite multiplier ".", patatom is satisfied by a concatenation of any number of *Si* (including none), each of which meets the specification of patatom.

If repcount has the form of a single intlit, patatom is satisfied by a concatenation of exactly intlit *Si*, each of which meets the specification of patatom. In particular, if the value of intlit is zero, the corresponding *Si* is empty.

If repcount has the form of a range, intlit1.intlit2, the first intlit gives the lower bound, and the second intlit the upper bound. If the upper bound is less than the lower bound an error condition occurs with ecode="M10". If the lower bound is omitted, so that the range has the form .intlit2 , the lower bound is taken to be zero. If the upper bound is omitted, so that the range has the form intlit1. , the upper bound is taken to be indefinite; that is, the range is at least intlit1 occurrences. Then patatom is satisfied by the concatenation of a number of *Si*, each of which meets the specification of patatom, where the number must be within the expressed or implied bounds of the specified range, inclusive.

The dual operator '? is defined by:

*A* '? *B* = '(*A* ? *B*)

**8 Statements**

**8.1 General statement rules**

Every statement starts with a statementword which dictates the syntax and interpretation of that statement instance. statementwords differing only in the use of corresponding upper and lower case letters are equivalent. The standard contains the following statementwords:

│ B[REAK] │

│ C[LOSE] │

│ D[O] │

│ E[LSE] │

│ │ EI │ │

│ │ ELSEIF │ │

│ F[OR] │

│ H[ALT] │

│ H[ANG] │

│ I[F] │

│ J[OB] │

│ K[ILL] │

│ L[OCK] │

│ M[ERGE] │

statementword ::= │ N[EW] │

│ O[PEN] │

│ Q[UIT] │

│ R[EAD] │

│ S[ET] │

│ TC[OMMIT] │

│ TRE[START] │

│ TRO[LLBACK] │

│ TS[TART] │

│ U[SE] │

│ V[IEW] │

│ W[RITE] │

│ X[ECUTE] │

│ Z[unspecified] │

Unused statementwords other than those starting with the letter "Z" are reserved for future enhancement of the standard.

Any implementation of the language must be able to recognize both the abbreviated statementword (i.e., the character(s) to the left of the "[" in the list above) and the full spelling of each statementword. When two statements have a common abbreviated statementword, their argument syntax uniquely distinguishes them.

The formal definition of the syntax of statement is a choice from among all of the individual statement syntax definitions of 8.2.

│ syntax of BREAK statement │

│ syntax of CLOSE statement │

│ . │

statement ::= │ . │

│ . │

│ syntax of XECUTE statement │

│ syntax of Z[unspecified] statement │

For all statements allowing multiple arguments, the form

statementword arg1, arg2, ... argn

is equivalent in execution to

statementword arg1 statementword arg2 ... statementword argn

Within a statement, all expratoms are evaluated in a left-to-right order with all expratoms that occur to the left of the expratom being evaluated, including the complete resolution of any indirection, prior to the evaluation of that expratom, except as explicitly noted elsewhere in this document. The expratom is formed by the longest sequence of characters that satisfies the definition of expratom. (See 7.1 for a description of expratom).

**8.1.1 Spaces in statements**

Spaces are not significant characters. The use of spaces is defined by the appropriate statement definition and subsections 6.2 Routine body, and 6.4 Embedded programs.

**8.1.2 Comment comment**

If a { character appears in the statementword initial-letter position, it is the start of a comment. The remainder of the characters to the next } character must consist of graphics only, but is otherwise ignored and nonfunctional.

**8.1.3 Statement argument indirection**

Indirection is available for evaluation of either individual statement arguments or contiguous sublists of statement arguments. The opportunities for indirection are shown in the syntax definitions accompanying the statement descriptions.

Typically, where a statementword carries an argument list, as in

statementword WS L argument

the argument syntax will be expressed as

│ individual argument syntax │

argument ::= │ │

│ @ expratom V L argument │

This formulation expresses the following properties of argument indirection.

a) Argument indirection may be used recursively.

b) A single instance of argument indirection may evaluate to one complete argument or to a sublist of complete arguments.

Unless the opposite is explicitly stated, the text of each statement specification describes the arguments **after** all indirection has been evaluated.

Unless expressed otherwise, if individual argument syntax allows the @ expratom contruct, then argument indirection has precedence, i.e., the restriction on the value of expratom comes from the V operator of the argument indirection, not any other type of indirection.

**8.1.4 Post conditional postcond**

All statements except ELSE, FOR, and IF may be made conditional as a whole by following the statementword immediately by the post-conditional postcond.

postcond ::= [ [ WS ] : [ WS ] tvexpr ]

If the postcond is absent or the postcond is present and the value of the tvexpr is true, the statement is executed. If the postcond is present and the value of the tvexpr is false, the statementword and its arguments are passed over without execution.

The postcond may also be used to conditionalize the arguments of DO and XECUTE. In such cases the arguments' expratoms that occur prior to the postcond are evaluated prior to the evaluation of the postcond.

**8.1.5 Statement timeout timeout**

The OPEN, LOCK, JOB, and READ statements employ an optional timeout specification, associated with the testing of an external condition, and a corresponding ELSE clause.

timeout ::= [ WS ] : [ WS ] numexpr

If the optional timeout is absent, the statement will proceed if the condition, associated with the definition of the statement, is satisfied; otherwise, it will wait until the condition is satisfied and then proceed.

If the optional timeout is present, the value of numexpr must be nonnegative. If it is negative, the value 0 is used. Numexpr denotes a *t*-second timeout, where *t* is the value of numexpr.

If *t* = 0, the condition is tested. If it is false, the ELSE clause is executed. Execution proceeds without delay.

If *t* is positive, execution is suspended until the condition is true, but in any case no longer than *t* seconds. If, at the time of resumption of execution, the condition is false, the ELSE clause is executed.

Note that in the case of statements with multiple arguments with timeouts, the ELSE clause is *not* executed until after all arguments have been executed, at which point it is executed only if the condition of the *last* argument with a timeout evaluated to false.

**8.1.6 Line reference lineref and Method reference methodref**

The $TEXT function contains in its arguments means for referring to particular lines within any routine. A reference to a line is either an entryref, a textref, or a labelref.

The DO and JOB statements, extrinsic functions, and extrinsic variables contain in their arguments means for referring to particular methods within any routine. A reference to a method is either an entryref or a labelref.

An entryref allows indirection of both the label and the routinename. A textref allows indirection and the specification of integer offsets from a label (eg, LOOP+5 references the fifth line after the line that has LOOP for a label). A labelref, on the other hand, allows neither label offsets nor indirection.

This subclause describes the means for making line and method references.

│ entryref │

lineref ::= │ textref │

│ labelref │

methodref ::= │ entryref │

│ labelref │

**8.1.6.1 Entry reference entryref**

The total line specification in DO and JOB is in the form of entryref.

│ dlabel [ ^ routineref ] │

entryref ::= │ │

│ ^ routineref │

│ label │

dlabel ::= │ │

│ @ expratom V dlabel │

│ [ VB environment VB ] routinename │

routineref ::= │ │

│ @ expratom V routineref │

When label is an instance of intlit, leading zeros are significant to its spelling. In any context, reference to a particular spelling of label which occurs more than once in a defining occurrence in the given routine will have undefined results. If the line reference (dlabel) is absent, the first line (for $TEXT) or method (for DO, JOB, exfunc, or exvar) is implied.

DO and JOB can refer to a method in a routine other than that in which they occur; this requires a means of specifying a routinename. If the routineref includes an environment, then the routine is fetched from the specified environment. Reference to a non-existent environment causes an error condition with an ecode="M26". If the routine reference (^ routineref) is absent, the routine being executed is implied.

**8.1.6.2 $TEXT reference textref**

The total line specification in $TEXT is in the form of textref.

│ dlabel [ + intexpr ] [ ^ routineref ] │

textref ::= │ │

│ ^ routineref │

Any line in a given routine may be denoted by mention of a label which occurs in a defining occurrence on or prior to the line in question. In any context, reference to a particular spelling of label which occurs more than once in a defining occurrence in the given routine will have undefined results. When label is an instance of intlit, leading zeros are significant to its spelling. If the line reference (dlabel [+intexpr]) is absent, the first line is implied.

If +intexpr is absent, the line denoted by dlabel is the one containing label in a defining occurrence. If +intexpr is present and has the value *n* '< 0, the line denoted is the *n*th line after the one containing label in a defining occurrence. A negative value of intexpr causes an error condition with ecode="M12".

The $TEXT function can refer to a line in a routine other than that in which they occur; this requires a means of specifying a routinename. If the routineref includes an environment, then the routine is fetched from the specified environment. Reference to a non-existent environment causes an error condition with an ecode="M26". If the routine reference (^ routineref) is absent, the routine being executed is implied.

**8.1.6.3 Label reference labelref**

When the DO or JOB statements or exfunc or exvar include parameters to be passed to the specified routine, the indirect forms of entryref are not permitted, and the specified method must have a formallist. The entry specification labelref is used instead:

│ label [ ^ [ VB environment VB ] routinename ] │

labelref ::= │ │

│ ^ [ VB environment VB ] routinename │

If the routine reference ( ^ [ | environment | ] routinename ) is absent, the routine being executed is implied. If the label is absent, the first method is implied. If the labelref includes an environment, then the routine is fetched from the specified environment. Reference to a non-existent environment causes an error condition with an ecode="M26".

In the context of a DO or JOB statement, an exfunc, or an exvar, a spelling of label which does not occur in a defining occurrence in the given routine causes an error condition with ecode="M13".

**8.1.6.4 External reference externref**

externref ::= & [ packagename . ] externalroutinename

packagename ::= name

externalroutinename ::= name [ ^ name ]

The ampersand (&) character designates a program whose namespace is external to the current Omega environment. The effects of passing parameters are as defined in 8.1.7 (Parameter Passing).

The packagename shall be from a namespace of those determined by the appropriate namespace registry. If packagename is not specified, implementors may, optionally, choose to provide a default package.

Bindings may have one or more namespaces; requirements to use these namespaces must be clearly stated in the specification of the binding. The term *package* is used herein to denote programs that are in possibly external environments. No implied one-to-one correspondence for all possible external packages exists.

The externalroutinename namespace is undefined; this is a function of a binding. Any external mapping between the externalroutinename and any name used by an external package is an implementation-specific issue. The externalroutinename shall be of the form name or name^name.

8.1.6.5 Library reference libraryref

libraryref ::= % libraryelement [ ^ library ]

libraryelement ::= name

library ::= name

If no library is specified as part of a libraryref then the libraries specified in ^$JOB($J,"LIBRARY") are used. Note: This does not imply that the libraries specified in ^$JOB($J,"LIBRARY") can necessarily be dynamically changed during the lifetime of a process.

Unless explicitly specified in an individual libraryelement definition accessing a libraryref has no effect on local variables for a process or $REFERENCE, except for a return value and changes to variables passed by reference.

If an argument to a libraryref has an invalid value (such as a value outside the domain of the function) the behavior of the reference to the libraryref is undefined.

The restrictions specified in 8.1.7 Parameter passing also apply to the referencing of libraryrefs.

If a libraryelement or a library is not available for a library reference then an error condition occurs with ecode = "M13".

**8.1.7 Parameter passing**

Parameter passing is a method of passing information in a controlled manner to and from a subroutine or process as the result of an exfunc, an exvar, or a DO statement with an actuallist, or to a process as the result of a JOB statement with an actuallist.

actuallist ::= ( [ L actual ] )

┌╴ ╶┐

│ . actualname │

actual ::= │ │

│ expr │

└╴ ╶┘

│ name │

actualname ::= │ │

│ @ expratom V actualname │

When parameter passing occurs, the method designated by the labelref must contain a formallist in which the number of names is greater than or equal to the number of actuals in the actuallist. The correspondence between actual and formallist name is defined such that the first actual in the actuallist corresponds to the first name in the formallist, the second actual corresponds to the second formallist name, etc. Similarly, the correspondence between the parameter list entries, as defined below, and the actual or formallist names is also by position in left-to-right order. If the syntax of actual is .actualname, then it is said that the actual is of the call-by-reference format; if the syntax of actual is expr it is said that the actual is of the call-by-value format; otherwise it is said that the actual is of the omitted-parameter format.

When parameter passing occurs, the following steps are executed:

a) Process the actuals in left-to-right order to obtain a list of DATA-CELL pointers called the parameter list. The parameter list contains one item per actual. The parameter list is created according to the following rules:

1) If the actual is call-by-value, then evaluate the expr and create a DATA-CELL with a zero tuple value equal to the result of the evaluation. The pointer to this DATA-CELL is the parameter list item.

2) If the actual is call-by-reference, search the NAME-TABLE for an entry containing the actual name. If an entry is found, the parameter list item is the DATA-CELL pointer in this NAME-TABLE entry. If the actual name is not found, create a NAME-TABLE entry containing the name and a pointer to a new (empty) DATA-CELL. This pointer is the parameter list item. If a jobargument contains a call-by-reference actual an error occurs with ecode="M40" .

3) If the actual is omitted-parameter, create a new (empty) DATA-CELL.

b) Place the information contained in the formallist in the PROCESS-STACK frame.

c) For each name in the formallist, search the NAME-TABLE for an entry containing the name and if the entry exists, copy the NAME-TABLE entry into the parameter frame and delete it from the NAME-TABLE. This step performs an implicit NEW on the formallist names.

d) For each item in the parameter list, create a NAME-TABLE entry containing the corresponding formallist name and the parameter list item (DATA-CELL pointer). This step binds the formallist names to their respective actuals.

As a result of these steps, two (or more) NAME-TABLE entries may point to the same DATA-CELL. As long as this common linkage is in effect, a SET or KILL of an lvn with one of the names appears to perform an implicit SET or KILL of an lvn with the other name(s). Note that a KILL does not undo this linkage of multiple names to the same DATA-CELL, although subsequent parameter passing or NEW statements may.

Execution is then initiated at the first statement in the method specified by the labelref. Execution of the subroutine continues until a CB is encountered or a QUIT is executed that is not within the scope of a subsequently executed statement. In the case of an exfunc or exvar, the method must be terminated by a QUIT with an argument.

At the time of the QUIT, the formallist names are unbound and the original saved values, including any undefined states, of the variables named in the formallist are restored. See 8.2.16 for a discussion of the semantics of the QUIT operation.

When calling to an externref, call-by-reference has the following additional implementation independent definition:

a) Upon return of control to M, changes to the value of the lvn referenced by the actualname shall be as if the lvn was modified by a SET statement. The exact mechanism performing this operation is unspecified.

b) The resultant events are unspecified, if the data in the Omega environment is modified while an external routine call is being made that references the modified data.

c) Local variables (see 7.1.1 Variables) that are not passed as parameters, will not necessarily be available to the external environment.

**8.1.8 User-defined mnemonicspaces**

When a controlmnemonic is used for a device which has a user-defined mnemonicspace (see 8.2.16) then the usage of the controlmnemonic in a READ and WRITE statement format in the form

/controlmnemonic(expr,...)

is computationally equivalent, with the exception of the effect on the naked indicator, to

DO *label*^*routine*(expr,...)

where *routine* is the user-defined mnemonicspace routine and *label* is controlmnemonic, unless controlmnemonic commences with a ? in which case it is replaced by %. If the controlmnemonics of the mnemonicspace are case-insensitive then *label* is controlmnemonic converted to upper-case. Unless specifically stated otherwise mnemonicspaces are case-sensitive.

The naked indicator is restored to its value prior to the execution of the controlmnemonic associated routine.

Any reference to a controlmnemonic within a user-defined mnemonicspace for which there is no associated method causes an error condition with ecode = M32.

│ CLOSE│

devicestatement ::= │ OPEN │

│ USE │

If a label of the form %*statement*, where *statement* is a devicestatement, exists in a mnemonicspace statement routine then execution of a statement which is a devicestatement with at least one deviceparam is computationally equivalent to

NEW KEYWORD,ATTRIB,I

SET (KEYWORD,ATTRIB)=*no*

FOR I=1:1:*no* DO

. SET KEYWORD(I)=*key*I

. IF $D(*att*I) SET ATTRIB(I)=*att*I

DO %*label*^*routine*(expr,.KEYWORD,.ATTRIB,*time*)

where *label* is the statementword converted to upper-case and expanded to the fully spelled out devicestatement, *routine* is the user-defined mnemonicspace statement routine, *no* is the number of deviceparams, KEYWORD and ATTRIB contain the individual deviceparams in deviceparameters fully evaluated with *keyi*=devicekeywordi or deviceattributei as appropriate and *atti*=expri if deviceparam is in the deviceattribute form, and *time* is absent of the evaluated expression from timeout if timeout is present.

The usage of the deviceparam form expr is implementation specific.

Any action implied by the presence of a mnemonicspace in such a statement takes effect before the above code is executed.

The labels %READ and %WRITE within a user-defined mnemonicspace statement routine are reserved for future enhancement of this standard.

During the execution of any user-defined mnemonicspace routine the effect of user-defined processing of controlmnemonics and statements for the same mnemonicspace is unspecified.

Note: $STORAGE may be affected by the execution of user-defined mnemonicspace code.

Note: It is the responsibility of the user-defined mnemonicspace routine to process the deviceparameters in the appropriate order and return $T appropriately in the event that a timeout is present.

**8.2 Statement definitions**

The specifications of all statements follow.

**8.2.1 BREAK**

B[REAK] postcond │ argument syntax unspecified │

BREAK provides an access point within the standard for nonstandard programming aids. BREAK without arguments suspends execution until receipt of a signal, not specified here, from a device.

**8.2.2 CLOSE**

C[LOSE] postcond WS L closeargument

│ devn [ [ WS ] : [ WS ] deviceparameters ] │

closeargument ::= │ │

│ @ expratom V L closeargument │

devn ::= [ | environment | ] expr

│ deviceparam │

deviceparameters ::= │ │

│ ( [ [ deviceparam ] : ] ... deviceparam ) │

deviceparam ::= │ expr │

│ devicekeyword │

│ deviceattribute = expr │

devicekeyword ::= name

deviceattribute ::= name

The order of execution of deviceparams is from left to right within a deviceparameters usage.

If there is no mnemonicspace in use for a device or the current mnemonicspace is the empty string then the implementation may allow any of the forms of deviceparam. The expr form may not be mixed with the other forms within the same deviceparameters.

In all other cases the expr form is not allowed.

devn identifies a device. (In this paragraph, *device* encompasses I/O devices, files, data sets, and other objects supporting OPEN, USE, READ, WRITE, and CLOSE statements.) When environment is omitted, the value of expr denotes one device. When environment is present, the value of environment denotes one set of devices, while the value of expr denotes one member of the set. The interpretation of the values is left to the implementor. Reference to a non-existent environment causes an error condition with an ecode = "M26".

The deviceparameters may be used to specify termination procedures or other information associated with relinquishing ownership, in accordance with implementor interpretation.

Each designated device is released from ownership. If a device is not owned at the time that it is named in an argument of an executed CLOSE, the statement has no effect upon the ownership and the values of the associated parameters of that device. Device parameters in effect at the time of the execution of CLOSE are retained for possible future use in connection with the device to which they apply. If the current device is named in an argument of an executed CLOSE, $IO is given a value of the empty string.

**8.2.3 DO**

│ block │

D[O] postcond WS │ │

│ L doargument │

│ entryref postcond │

│ │

doargument ::= │ labelref actuallist postcond │

│ │

│ externref [ actuallist ] postcond │

│ │

│ @ expratom V L doargument │

A DO block initiates execution of its dependent block. If postcond is present and its tvexpr is false, the execution of the statement is complete. If postcond is absent, or the postcond is present and its tvexpr is true, the DO places a block frame containing the current execution location on the PROCESS-STACK, and continues execution at the first codeitem in the block. (See 6.3 for an explanation of routine execution.) When encountering an implicit or explicit QUIT not within the scope of a subsequently executed doargument, DO block, xargument, exfunc, exvar, or FOR, execution of this block is terminated (see 8.2.16 for a description of the actions of QUIT). Execution resumes at the statement (if any) following the DO block.

DO with arguments is a generalized call to the method specified by the entryref, the labelref, or the externref in each doargument.

If the actuallist is present in an executed doargument, parameter passing occurs and the method designated by labelref must contain a formallist in which the number of names is greater than or equal to the number of actuals in the actuallist. If the call is to an externref and an actuallist is present, then parameter passing occurs, and data is transferred (with any conversion as defined in the binding to the external package).

Each doargument is executed, one at a time in left-to-right order, in the following steps.

a) Evaluate the expratoms of the doargument.

b) If postcond is present and its tvexpr is false, execution of the doargument is complete. If postcond is absent, or postcond is present and its tvexpr is true, proceed to the step c.

c) A DO-frame containing the current execution location is placed on the PROCESS-STACK.

d) If the actuallist is present, execute the sequence of steps described in 8.1.7 Parameter Passing.

e) Continue execution at the first statement position specified by the reference as follows:

1) For entryref and labelref, this is the first statement in the dependent block of the method specified by entryref or labelref. Execution of the method (within the Omega environment) continues until an implicit or explicit QUIT is executed that is not within the scope of a subsequently executed FOR, DO block, doargument, xargument, exfunc, or exvar. The scope of this internally referenced doargument is said to extend to the execution of that QUIT. (See 8.2.16 for a description of the actions of QUIT.) Execution then returns to the first character position following the doargument.

2) For externref, this is the first executable item as specified within the package environment. If the reference is external to M, execution proceeds in the specified environment until termination, as defined within that environment, occurs. Execution then returns to the first character following the doargument.

**8.2.5 FOR**

F[OR] │ [ WS ] │ block

│ │

│ WS lvn = L forparameter │

│ expr │

forparameter ::= │ numexpr1 : numexpr2 : numexpr3 │

│ numexpr1 : numexpr2 │

The FOR statement performs iteration of a block of code. The *scope* of the FOR statement is its dependent block.

The FOR with arguments specifies repeated execution of the statements within its scope for different values of the local variable lvn, under successive control of the forparameters, from left to right. Any expressions occurring in lvn, such as might occur in subscripts or indirection, are evaluated once per execution of the FOR, prior to the first execution of any forparameter.

For each forparameter, control of the execution of the statements in the scope is specified as follows. (Note that *A*, *B*, and *C* are hidden temporaries.)

a) If the forparameter is of the form expr1.

1) Set lvn = expr.

2) Execute the statements in the scope once.

3) Processing of this forparameter is complete.

b) If the forparameter is of the form numexpr1 : numexpr2 : numexpr3 and numexpr2 is nonnegative.

1) Set *A* = numexpr1.

2) Set *B* = numexpr2.

3) Set *C* = numexpr3.

4) Set lvn = *A*.

5) If lvn > *C*, processing of this forparameter is complete.

6) Execute the statements in the scope once.

7) If lvn > *C*−*B*, processing of this forparameter is complete; an undefined value for lvn causes an error condition with ecode="M15".

8) Otherwise, set lvn = lvn + *B*.

9) Go to 6.

c) If the forparameter is of the form numexpr1 : numexpr2 : numexpr3 and numexpr2 is negative.

1) Set *A* = numexpr1.

2) Set *B* = numexpr2.

3) Set *C* = numexpr3.

4) Set lvn = *A*.

5) If lvn < *C*, processing of this forparameter is complete.

6) Execute the statements in the scope once.

7) If lvn < *C*−*B*, processing of this forparameter is complete; an undefined value for lvn causes an error condition with ecode="M15".

8) Otherwise, set lvn = lvn + *B*.

9) Go to 6.

d) If the forparameter is of the form numexpr1 : numexpr2.

1) Set *A* = numexpr1.

2) Set *B* = numexpr2.

3) Set lvn = *A*.

4) Execute the statements in the scope once.

5) Set lvn = lvn + *B*; an undefined value for lvn causes an error condition with ecode="M15".

6) Go to 4.

If the FOR statement has no argument:

a) Execute the statements in the FOR block once; since no lvn has been specified, it cannot be referenced.

b) Goto a.

Note that if the FOR block (the *outer* FOR) contains an *inner* FOR, one execution of the outer FOR block encompasses all executions of the inner FOR block corresponding to one complete pass through the inner FOR statement's forparameter list.

Note that form d, and the argumentless FOR, specify endless loops. Termination of these loops must occur by execution of an argumentless QUIT within the FOR block. This method is available within the FOR block independent of the form of forparameter currently in control of its execution; it is described below. Note also that no forparameter to the right of one of form d can be executed.

Execution of an argumentless QUIT within a FOR block has two effects.

a) It terminates that particular execution of the block at the QUIT; statements within the FOR block subsequent to the QUIT are not executed.

b) It causes any remaining values of the forparameter in control at the time of execution of the QUIT, and the remainder of the forparameters in the same forparameter list, not to be calculated and the statements in the FOR block not to be executed under their control.

In other words, execution of the QUIT effects the immediate termination of the innermost FOR whose dependent block contains the QUIT. Note that the execution of the QUIT within a FOR block does not affect the variable environment.

Execution of an argumented QUIT within the scope of a FOR statement causes an error condition with an ecode="M16".

**8.2.6 HALT**

H[ALT] postcond [ WS ]

If the value of $TLEVEL is greater then zero, a ROLLBACK is performed. In any case, all nrefs are removed from the LOCK-LIST associated with this process. Finally, execution of this process is terminated.

**8.2.7 HANG**

H[ANG] postcond WS L hangargument

│ numexpr │

hangargument ::= │ │

│ @ expratom V L hangargument │

Let *t* be the value of numexpr. If *t* '> 0, HANG has no effect. Otherwise, execution is suspended for *t* seconds.

**8.2.8 IF**

I[F] WS L ifargument [ WS block [ WS elseif ] ... [ WS else ] ]

│ tvexpr │

ifargument ::= │ │

│ @ expratom V L ifargument │

elseif ::= │ EI │ WS L ifargument WS block

│ ELSEIF │

else ::= E[LSE] WS block

The IF statement tests conditions and selects at most a single block to execute based on the results. It executes from left to right a sequence of clauses, each of which is comprised of a statementword, a list of ifarguments (empty in the case of an ELSE clause), and an associated block. If an executed clause's condition holds true, its associated block is executed, and the IF statement ends; otherwise, execution passes to the next clause. Execution of the IF statement ends if all clauses are tested without finding one whose condition holds true.

An IF or ELSEIF clause holds true if the value of each of its ifarguments evaluates to 1. Note that multiple ifarguments are executed in sequence, such that if any tvexpr evaluates to 0, subsequent tvexprs are *not* evaluated.

An ELSE clause always holds true.

If the IF statement contains no IF block, the block containing the IF statement terminates if any of the ifarguments evaluates to 0; otherwise, execution continues with the next statement after the IF.

**8.2.9 JOB**

J[OB] postcond WS L jobargument [ else ]

│ [ jobenv ] [ : jobparameters ] │

jobargument ::= │ │

│ [ jobenv ] labelref actuallist [ : jobparameters ] │

│ │

│ @ expratom V L jobargument │

jobenv ::= VB environment VB

│ processparameters [ timeout ] │

jobparameters ::= │ │

│ timeout │

│ expr │

processparameters ::= │ │

│ ( [ [ expr ] : ] ... expr ) │

For each jobargument, the JOB statement attempts to initiate another Omega process. If the actuallist is present in a jobargument, the method designated by labelref must contain a formallist in which the number of names is greater than or equal to the number of exprs in the actuallist.

The JOB statement initiates this process at the method specified by the entryref or labelref. There is no linkage between the started process and the process that initiated it. It is erroneous for a jobargument to contain a call-by-reference actual (ecode="M40"). If the actuallist is not present, the process will have no variables initially defined. (See 7.1.2.3 Process-Stack, and 8.1.7 Parameter passing).

The processparameters can be used in an implementation-specific fashion to indicate partition size, principal device, and the like.

If no timeout is present, process execution is suspended until the process named in the jobargument is successfully initiated, and the ELSE clause is not executed. If a timeout is present, it determines the maximum time to wait until the process named in the jobargument is successfully initiated; after all jobarguments have executed, if the last argument with a timeout did not successfully initiate its process, the ELSE block is executed. The meaning of success in either context is defined by the implementation.

If jobenv is explicitly specified, the JOB statement attempts to initiate this process in the environment specified by jobenv. Reference to a non-existent jobenv causes an error condition with an ecode = "M26". If jobenv is not explicitly specified, then the value of ^$JOB($JOB,"JOB") is used.

**8.2.10 KILL**

K[ILL] postcond killarglist

│ [ WS ] │

killarglist ::= │ │

│ WS L killargument │

│ glvn │

killargument ::= │ ( L lname ) │

│ @ expratom V L killargument │

│ name │

lname ::= │ │

│ @ expratom V lname │

The three forms of KILL are given the following names.

a) glvn: Selective Kill.

b) (L lname): Exclusive Kill.

c) Empty argument list: Kill All.

KILL is defined using a subsidiary function K(*V,val,subs*) where *V* is a glvn, *val*  is 1, and *subs*  is 1.

a) Search for the name of *V* in the NAME-TABLE. If no such entry is found, the function is completed. Otherwise, extract the DATA-CELL pointer and proceed to step b.

b) In the DATA-CELL identified in step 'a', let N be the number of subscripts in *V*. If *V* is unsubscripted, let N be 0:

1) If N is 0, then delete all tuples. The function is completed.

2) Otherwise (if N > 0), delete all tuples of degree N or greater whose first N subscripts are the same as those in *V*. The function is completed.

Note that as a result of procedure *K*(*V*,1,1), $DATA(*V*)=0, i.e., the value of *V* is undefined, and *V* has no descendants.

The actions of the three forms of KILL are then defined as:

a) Selective Kill Apply procedure *K*(glvn,1,1).

b) Exclusive Kill For all names, *V*, in the local variable NAME-TABLE except those in the argument list, apply procedure *K*(glvn,1,1). Note that the names in the argument list of an exclusive kill are restricted to unsubscripted locals.

c) Kill All For all names, *V*, in the local variable NAME-TABLE, apply procedure *K*(glvn,1,1). Note that Kill All applies procedure *K* to the local variable NAME-TABLE only.

If a variable *N*, a descendant of *M*, is killed, the killing of *N* affects the value of $DATA(*M*) as follows: if *N* was not the only descendant of *M*, $DATA(*M*) is unchanged; otherwise, if *M* has a defined value $DATA(*M*) is changed from 11 to 1; if *M* does not have a defined value $DATA(*M*) is changed from 10 to 0.

**8.2.11 KSUBSCRIPTS**

KS[UBSCRIPTS] postcond killarglist

The three forms of KSUBSCRIPTS are given the following names.

a) glvn: Selective Kill.

b) (L lname): Exclusive Kill.

c) Empty argument list: Kill All.

KSUBSCRIPTS is defined using a subsidiary function K(*V,val,subs*) where *V* is a glvn, *val* is 0, and *subs* is 1.

a) Search for the name of *V* in the NAME-TABLE. If no such entry is found, the function is completed. Otherwise, extract the DATA-CELL pointer and proceed to step b.

b) In the DATA-CELL identified in step 'a', let N be the number of subscripts in *V*. If *V* is unsubscripted, let N be 0. Delete all tuples of degree N+1 or greater whose first N subscripts are the same as those in *V*. The function is completed.

Note that as a result of procedure *K*(*V*,0,1), $D(*V*)=1 if *V* had a value before procedure *K* was applied, or $D(*V*)=0 if *V* had no value before procedure *K* was applied, i.e., only the descendants of *V* are deleted.

The actions of the three forms of KILL are then defined as:

a) Selective Kill Apply procedure *K*(glvn,0,1).

b) Exclusive Kill For all names, *V*, in the local variable NAME-TABLE except those in the argument list, apply procedure *K*(glvn,0,1). Note that the names in the argument list of an exclusive kill are restricted to unsubscripted locals.

c) Kill All For all names, *V*, in the local variable NAME-TABLE, apply procedure *K*(glvn,0,1). Note that Kill All applies procedure *K* to the local variable NAME-TABLE only.

If a variable *N*, a descendant of *M*, is killed, the killing of *N* affects the value of $DATA(*M*) as follows: if *N* was not the only descendant of *M*, $DATA(*M*) is unchanged; otherwise, if *M* has a defined value $DATA(*M*) is changed from 11 to 1; if *M* does not have a defined value $DATA(*M*) is changed from 10 to 0.

**8.2.12 KVALUE**

KV[ALUE] postcond killarglist

The three argument forms of KVALUE are given the following names.

a) glvn: Selective Kill.

b) (L lname): Exclusive Kill.

c) Empty argument list: Kill All.

KVALUE is defined using a subsidiary function K(*V,val,subs*) where *V* is a glvn, *val* is 1, and *subs* is 0.

a) Search for the name of *V* in the NAME-TABLE. If no such entry is found, the function is completed. Otherwise, extract the DATA-CELL pointer and proceed to step b.

b) If *val*=1 and *subs*=0 then in the DATA-CELL identified in step 'a':

1) If *V* is unsubscripted, delete the tuple of degree 0 (if found). The function is completed.

2) Otherwise, let N be the number of subscripts in *V*. Delete (if found) only the tuple of degree whose first N subscripts are the same as those in *V*. The function is completed.

Note that as a result of procedure *K*(*V*,1,0), $D(*V*)=0 if *V* had no descendants before procedure *K* was applied, or $D(*V*)=10 if *V* had descendants before procedure *K* was applied, i.e., only the value of *V* is deleted.

The actions of the three forms KVALUE are then defined as:

a) Selective Kill Apply procedure *K*(glvn,1,0).

b) Exclusive Kill For all names, *V*, in the local variable NAME-TABLE except those in the argument list, apply procedure *K*(glvn,1,0). Note that the names in the argument list of an exclusive kill are restricted to unsubscripted locals.

c) Kill All For all names, *V*, in the local variable NAME-TABLE, apply procedure *K*(glvn,1,0). Note that Kill All applies procedure *K* to the local variable NAME-TABLE only.

If a variable *N*, a descendant of *M*, is killed, the killing of *N* affects the value of $DATA(*M*) as follows: if *N* was not the only descendant of *M*, $DATA(*M*) is unchanged; otherwise, if *M* has a defined value $DATA(*M*) is changed from 11 to 1; if *M* does not have a defined value $DATA(*M*) is changed from 10 to 0.

**8.2.13 LOCK**

│ [ WS ] │

L[OCK] postcond │ │

│ WS L lockargument [ else ] │

┌─ ─┐

│ │ + │ │ nref │ │

│ │ │ │ │ [ timeout ] │

lockargument ::= │ │ - │ │ ( L nref ) │ │

│ └─ ─┘ │

│ @ expratom V L lockargument │

│ [ ^ ] [ VB environment VB ] name [ ( L expr ) ] │

nref ::= │ │

│ @ expratom V nref │

LOCK provides a generalized interlock facility available to concurrently executing Omega processes to be used as appropriate to the applications being programmed. Execution of LOCK is not affected by, nor does it directly affect, the state or value of any global or local variable, or the value of the naked indicator. Its use is not required to access globals, nor does its use inhibit other processes from accessing globals. It is an interlocking mechanism whose use depends on programmers establishing and following conventions.

Each lockargument specifies a subspace of the total Omega LOCK-UNIVERSE for the environment upon which the executing process seeks to make or release an exclusive claim; the details of this subspace specification are given below.

A special space for the lockspace is needed to create a synchronization mechanism for the executing process for each of the environments referenced by the executing process. A timeout refers to the time spent at the target environment, any time delays due to communication delays are not part of the timeout.

For the purposes of this discussion, the LOCK-UNIVERSE is defined as the union of all possible nrefs in one environment after resolution of all indirection. Further, there exists for each process a LOCK-LIST that contains zero or more nrefs. Execution of lockarguments has the effect of adding or removing nrefs from the process' LOCK-LIST. A given nref may appear more than once within the LOCK-LIST. The nrefs in the LOCK-LIST specify a subset of the LOCK-UNIVERSE. This subspace, called the process' LOCKSPACE, consists of the union of the subspaces specified by all nrefs in the LOCK-LIST, as follows:

a) If the nref is unsubscripted, then the subspace is the set of the following points: one point for the unsubscripted variable name nref and one point for each subscripted variable name *N*(*s*1,...,*si*) where *N* has the same spelling as nref.

b) If the occurrence of nref is subscripted, let the nref be *N*(*s*1,*s*2,...,*sn*). Then the subspace is the set of the following points: one point for *N(s1,s2,...,sn)* and one point for each descendant (see 7.1.5.3 $DATA function for a definition of descendant) of nref.

If the LOCK statement is argumentless, LOCK removes all nrefs from the LOCK-LIST associated with this process.

Execution of lockargument occurs in the following order:

a) Any expression evaluation involved in processing the lockargument is performed.

b) If the form of lockargument does not include an initial + or − sign, then prior to evaluating or executing the rest of the lockargument, LOCK first removes all nrefs from the LOCK-LIST associated with this process. Then it appends each of the nrefs in the lockargument to the process' LOCK-LIST.

c) If the lockargument has a leading + sign, LOCK appends each of the nrefs in the lockargument to the process' LOCK-LIST.

d) If the lockargument has a leading − sign, then for each nref in the lockargument, if the nref exists in the LOCK-LIST for this process, one instance of nref is removed from the LOCK-LIST.

An error occurs, with ecode="M41", if a process within a TRANSACTION attempts to remove from its LOCK-LIST any nref that was present when the TRANSACTION started. With respect to each other process, the effect of removing any nref from the LOCK-LIST is deferred until the global variable modifications made since that nref was added to the LOCK-LIST are available to that other process.

LOCK affects concurrent execution of processes having LOCK-SPACES that OVERLAP. Two LOCK-SPACEs OVERLAP when their intersection is not empty. LOCK imposes the following constraints on the concurrent execution of processes:

a) The LOCK-SPACEs of any two processes executing statements outside the scope of a TRANSACTION may not OVERLAP.

b) All global variable modifications produced by the execution of statements by processes having LOCK-SPACEs that OVERLAP must be equivalent to the modifications resulting from some execution schedule during which their LOCK-SPACEs do not OVERLAP.

See the TRANSACTION Processing subclause for the definition of TRANSACTION.

The constraints imposed by LOCK on the execution of processes having LOCK-SPACEs that OVERLAP may cause execution of one or more processes to be delayed. The maximum duration of such a delay may be specified with a timeout.

If present, timeout modifies the execution of LOCK, described above, as follows:

a) If execution of the process is delayed and cannot be resumed prior to the expiration of timeout, then the execution of the lockargument is unsuccessful. In this event any nrefs added to the LOCK-LIST as a result of executing the lockargument are removed.

b) Otherwise, the execution of the lockargument is successful.

After all lockarguments have executed, if the last argument with a timeout did not successfully establish its lock, the ELSE block is executed.

**8.2.14 MERGE**

M[ERGE] postcond WS L mergeargument

╷ ╷

│ glvn1 = glvn2 │

mergeargument ::= │ │

│ @ expratom V L mergeargument │

MERGE provides a facility to copy a glvn2 into a glvn1 and all descendants of glvn2 into descendants of glvn1 according to the scheme described below.

MERGE does not KILL any nodes in glvn1, or any of its descendants.

Assume that glvn1 is represented as A(i1, i2, ..., ix) (x'<0) and that glvn2 is represented as B(j1, j2, ..., jy) (y'<0).

Then:

a) If $DATA(B(j1,j2,...,jy)) has a value of 1 or 11, then the value of glvn2 is given to glvn1.

b) The value for every occurrence of z, such that z > 0 and $DATA(B(j1, j2,...,jy+z)) has a value of 1 or 11, the value of B(j1,j2,...,jy+z) is given to A(i1, i2,...,ix,jy+1,jy+2,...,jy+z).

The state of the naked indicator will be modified as if $DATA(glvn2)#10=1 and the statement SET glvn1=glvn2 would have been executed.

If glvn1 is a descendant of glvn2 or if glvn2 is a descendant of glvn1 an error condition occurs with ecode="M19".

**8.2.15 NEW**

│ [ WS ] │

N[EW] postcond │ │

│ WS L newargument │

│ lname │

│ newsvn │

newargument ::= │ ( L lname ) │

│ @ expratom V L newargument │

newsvn ::= │ $ET[RAP] │

│ $ES[TACK] │

NEW provides a means of performing variable scoping.

The three argument forms of NEW are given the following names:

a) lname: Selective NEW

b) (L lname): Exclusive NEW

c) Empty argument list: NEW All

d) newsvn NEW svn

The following discussion uses terms defined in the Variable Handling (see 7.1.2.2) and Process-Stack (see 7.1.2.3) models and, like those subclauses, does not imply a required implementation technique. Each argument of the NEW statement creates a CONTEXT-STRUCTURE consisting of a NEW NAME-TABLE and an exclusive indicator, attaches it to a linked list of CONTEXT-STRUCTUREs associated with the current PROCESS-STACK frame, and modifies currently active NAME-TABLEs as follows:

a) NEW All marks the CONTEXT-STRUCTURE as exclusive, copies the currently active NAME-TABLE to the NEW NAME-TABLE and makes all entries in the currently active local variable NAME-TABLE point to empty DATA-CELLs.

b) Exclusive NEW marks the CONTEXT-STRUCTURE as exclusive, copies the currently active NAME-TABLE to the NEW NAME-TABLE and changes all entries in the currently active local variable NAME-TABLE, except for those corresponding to names specified by the statement argument, to point to empty DATA-CELLs.

c) Selective NEW copies the entry corresponding to the name specified by the statement argument to the NEW NAME-TABLE and makes that entry in the currently active NAME-TABLE point to an empty DATA-CELL.

d) NEW svn copies the entry corresponding to the name specified by the statement argument to the NEW NAME-TABLE and updates that entry as follows:

1) if the argument specifies $ESTACK, points to a DATA-CELL with a value of 0 (zero).

2) if the argument specifies $ETRAP, points to a DATA-CELL with a value copied from the prior DATA-CELL (as pointed to by the just-copied NAME-TABLE entry).

**8.2.16 OPEN**

O[PEN] postcond WS L openargument [ else ]

│ devn [ : openparameters ] │

openargument ::= │ │

│ @ expratom V L openargument │

│ deviceparameters [ timeout [ : mnemonicspec ] ] │

│ │

openparameters ::= │ [ deviceparameters ] :: mnemonicspec │

│ │

│ timeout [ : mnemonicspec ] │

│ mnemonicspace │

mnemonicspec ::= │ │

│ ( L mnemonicspace ) │

mnemonicspace ::= expr V mnemonicspacename

│ ┌─╴ ╶──┐ │

│ │ ident │ │

mnemonicspacename ::= │ ident │ digit │ ... │

│ │ . │ │

│ │ - │ *(Note: hyphen)* │

│ └─╴ ╶─┘ │

│ ^ routineref1 [ ^ routineref2 ] │

mnemonicspace specifies the set of controlmnemonics that may be used within format arguments to subsequent READ and WRITE statements. The mnemonicspace may be an empty string and may not provide any defined controlmnemonics. mnemonicspacenames that start with any character other than "Y" or "Z" are reserved for mnemonicspace definitions registered by the MDC; those that start with "Z" are implementor-specific.

The ^routineref alternative is a user-defined mnemonicspace and associates the routine named in routineref1 with the location of code to be executed when a controlmnemonic is used.

The user-defined mnemonispace statement routine is the routine defined in routineref2, or if absent in routineref1. It associates this routine with the location of code to be executed when a statement is used in conjunction with the mnemonicspace.

If an implementation does not provide for the use of a specific mnemonicspace then that implementation shall provide a mechanism by which to associate a routineref with this mnemonicspace. All subsequent references to this mnemonicspace are handled as if this were a user-defined mnemonicspace.

When a mnemonicspec contains a list of mnemonicspaces, the first one determines the active mnemonicspace, which may be changed by a USE statement.

If the device does not support any mnemonicspace in a mnemonicspec, an error condition occurs with ecode = "M35". If any mnemonicspaces in the mnemonicspec are incompatible, an error occurs with ecode = "M36".

In addition to controlmnemonics a mnemonicspace also defines the valid deviceattributes and devicekeywords which are associated with a device. deviceattributes and devicekeywords which start with the character “Z” are implementor-specific. Associated with each deviceattribute are one or more values which are held in the ssvn ^$DEVICE.

See 8.2.2 for the syntax and interpretation of devn and deviceparameters.

The OPEN statement is used to obtain ownership of a device, and does not affect which device is the current device or the value of $IO. (see the discussion of USE in 8.2.25)

For each openargument, the OPEN statement attempts to seize exclusive ownership of the specified device. OPEN performs this function effectively instantaneously as far as other processes are concerned; otherwise, it has no effect regarding the ownership of devices and the values of the device parameters. If no timeout is present, process execution is suspended until seizure of ownership has been successfully accomplished by the process that issued the OPEN statement. After all openarguments have executed, if the last argument with a timeout did not successfully obtain ownership of its device, the ELSE block is executed.

Ownership is relinquished by execution of the CLOSE statement. When ownership is relinquished, all device parameters are retained. Upon establishing ownership of a device, any parameter for which no specification is present in the openparameters is given the value most recently used for that device; if none exists, an implementor-defined default value is used.

**8.2.17 QUIT**

│ [ WS ] │

│ │

Q[UIT] postcond │ WS expr │

│ │

│ WS @ expratom V expr │

QUIT terminates execution of DO block, doargument, xargument, exfunc, exvar, or FOR statement.

Encountering the end-of-routine mark eor; or encountering a CB at the end of a DO block, XECUTE block, or method block; is equivalent to an unconditional argumentless QUIT.

The effect of executing QUIT in the scope of FOR is fully discussed in 8.2.5. Note the eor never occurs in the scope of FOR.

If an executed QUIT is not in the scope of FOR, then it is in the scope of some DO block, doargument, xargument, exfunc, or exvar if not explicitly then implicitly, because the initial activation of a process, including that due to execution of a jobargument, may be thought of as arising from execution of a DO naming the first executed routine of that process.

The effect of executing a QUIT in the scope of a DO block, doargument, xargument, exfunc, or exvar is to restore the previous variable environment (if necessary) and continue execution at the location of the invoking DO block, doargument, xargument, exfunc, or exvar.

If the expr is present in the QUIT and the return is not to an exfunc or exvar, an error condition occurs with ecode="M16". If the expr is not present and the return is to an exfunc or exvar, an error condition occurs with ecode="M17".

The following discussion uses terms defined in the Variable Handling (see 7.1.2.2) and Process-Stack (see 7.1.2.3) models and, like those subclauses, does not imply a required implementation technique.

Execution of a QUIT occurs as follows:

a) If an expr is present, evaluate it. This value becomes the value of the invoking exfunc or exvar.

b) Remove the frame on the top of the PROCESS-STACK. If no such frame exists, then execute an implicit HALT.

c) If the PROCESS-STACK frame's linked list of CONTEXT-STRUCTUREs contains NEW NAME-TABLEs, process them in last-in-first-out order from their creation. If the CONTEXT-STRUCTURE is exclusive, make all entries in the currently active local variable NAME-TABLE point to empty DATA-CELLs. In all cases, the NEW NAME-TABLEs are copied to the currently active NAME-TABLEs. Note that, in the model, QUIT never encounters any restart CONTEXT-STRUCTUREs in the linked list because they must have been removed by TCOMMITs or ROLLBACKs for the QUIT to reach this point in its execution.

d) If the frame contains formal list information, extract the formallist and process each name in the list with the following steps:

1) Search the NAME-TABLE for an entry containing the name. If no such entry is found, processing of this name is complete. Otherwise, proceed to step 2.

2) Delete the NAME-TABLE entry for this name.

3) Finally, copy all NAME-TABLE entries from this frame into the NAME-TABLE.

4) Processing of this frame is complete, continue at step b.

e) If the frame is a TSTART frame and $TLEVEL is greater than zero, QUIT generates an error with ecode="M42". If the frame is a TSTART frame and $TLEVEL is zero, then the frame is discarded.

f) Continue execution at the location specified in the frame.

**8.2.18 READ**

R[EAD] postcond WS L readargument [ else ]

│ strlit │

│ format │

readargument ::= │ glvn [ readcount ] [ timeout ] │

│ \* glvn [ timeout ] │

│ @ expratom V L readargument │

readcount ::= # intexpr

The readarguments are executed, one at a time, in left-to-right order.

The forms strlit and format cause output operations to the current device; the forms glvn and \*glvn cause input from the current device to the named variable (see 7.1.2.4 for a description of the value assignment operation). If no timeout is present, execution will be suspended until the input message is terminated, either explicitly or implicitly with a readcount. (See 8.2.25 for a definition of *current device*.)

If a timeout is present, it is interpreted as a *t*-second timeout, and execution will be suspended until the input message is terminated, but in any case no longer than *t* seconds. If *t* '> 0, *t* = 0 is used.

After all lockarguments have executed, if the input message of the last argument with a timeout is not terminated at or before the time at which execution resumes, the ELSE block is executed.

When the form of the argument is \*glvn [ timeout ], the input message is by definition one character long, and it is explicitly terminated by the entry of one character, which is not necessarily from the ASCII set. The value given to glvn is an integer; the mapping between the set of input characters and the set of integer values given to glvn may be defined by the implementor in a device-dependent manner. If timeout is present and the timeout expires, glvn is given the value −1.

When the form of the argument is glvn [ timeout ], the input message is a string of arbitrary length which is terminated by an implementor-defined procedure, which may be device-dependent. If timeout is present and the timeout expires, the value given to glvn is the string entered prior to expiration of the timeout; otherwise, the value given to glvn is the entire string.

When the form of the argument is glvn # intexpr [ timeout ], let *n* be the value of intexpr.  *If n* '> 0 an error condition occurs with ecode="M18". Otherwise, the input message is a string whose length is at most *n* characters, and which is terminated by an implementor-defined, possibly device-dependent procedure, which may be the receipt of the *n*th character. If timeout is present and the timeout expires prior to the termination of the input message by either mechanism just described, the value given to glvn is the string entered prior to the expiration of the timeout; otherwise, the value given to glvn is the string just described.

When it has been specified that the current device is able to send control-sequences according to some mnemonicspace, the READ will be terminated as soon as such a control-sequence has been entered (be it by typing a function-key or by some other internal process within the device). The value of the specified glvn will be the same as if instead of the control-sequence the usual terminator-character would have been received before the control-sequence was sent.

When the form of the argument is strlit, it is equivalent to WRITE strlit. When the form of the argument is format, it is equivalent to WRITE format.

$X and $Y are affected by READ the same as if the statement were WRITE with the same argument list (except for timeouts and readcounts) and with each expr value in each writeargument equal, in turn, to the final value of the respective glvn resulting from the READ.

Input operations, except when the form of the argument is \*glvn [ timeout ], are affected by the Character Set Profile input-transform. Output operations are affected by the Character Set Profile output-transform. (see 7.1.3.1 ^$CHARACTER)

**8.2.19 RLOAD**

RL[OAD] postcond WS L routineargument

│ routineref : glvn [ : routineparameters ] │

routineargument ::= │ │

│ @ expratom V L routineargument │

│ routineparam │

routineparameters ::= │ │

│ ( [ [ routineparam ] : ] ... routineparam ) │

│ routinekeyword │

routineparam ::= │ │

│ routineattribute = expr │

routinekeyword ::= name

routineattribute ::= name

Spellings of routinekeyword and routineattribute differing only in the use of lowercase and uppercase letters are equivalent. All values of routinekeyword and routineattribute not starting with the character ‘Z’ are reserved for the MDC.

Assume that glvn is represented as A(i1,i2,...,ix) (x’<0). Then the lines of the routine denoted by routineref are stored in nodes A(i1,i2,...,ix,ix+1). ix+1 has a value of n for the nth line of the routine for all lines of the routine, and no other nodes of A within the subscript range i1..ix+1 will be affected.

The naked indicator is modified by the reference to glvn if it is a gvn, but not by the implicit reference to the immediate descendants of glvn.

If the routineref denotes a non-existent routine an error condition occurs with an ecode=”M88".

**8.2.20 RSAVE**

RS[AVE] postcond WS L routineargument

Assume that glvn is represented as A(i1,i2,...ix) (x'<0). Then the data values of all nodes A(i1,i2,...,ix,ix+1) for which the value of $DATA is either 1 or 11 are stored as lines of the routine denoted by routineref. The lines are taken in the subscript ordering for ix+1 as specified in the definition of $ORDER (7.1.5.11).

If glvn is undefined or if no node A(i1,i2,...,ix,ix+1) with a $DATA value of 1 or 11 exists the routine denoted by routineref is deleted.

If any one of the lines denoted by A(i1,i2,...,ix,ix+1) does not conform to the definition of a line the effect of executing the RSAVE statement is unspecified.

At no point during the execution of the RSAVE statement will any process be able to see a partially-filed routine.

Execution of a RSAVE statement where routineref names the currently-executing routine causes an error with ecode="M25", and the routine is not modified.

The naked indicator is modified by the reference to glvn if it is a glvn, but not by the implicit reference to the immediate descendants of glvn.

**8.2.21 SET**

S[ET] postcond WS L setargument

setargument ::= ╷ setdestination = expr ╷

│ @ expratom V L setargument │

setdestination ::= ╷ setleft ╷

│ ( L setleft ) │

╷ leftrestricted ╷

setleft ::= │ leftexpr │

│ glvn │

╷ $D[EVICE] ╷

│ $K[EY] │

leftrestricted ::= │ $R[EFERENCE]│

│ $X │

│ $Y │

╷ setpiece ╷

leftexpr ::= │ setextract │

│ setev │

│ setqsub │

setpiece ::= $P[IECE] ( glvn , expr1 [ , intexpr1 [ , intexpr2 ] ] )

setextract ::= $E[XTRACT] ( glvn [ , intexpr1 [ , intexpr2 ] ] )

setev ::= │ $EC[ODE] │

│ $ET[RAP] │

setqsub ::= $QS[UBSCRIPT] ( glvn , intexpr )

See 7.1.2 for the definition of glvn. See 7.1.4.6 for the definition of intexpr.

SET is the general means both for explicitly assigning values to variables, and for substituting new values in pieces of a variable. Each setargument computes one value, defined by its expr. That value is then either assigned to each of one or more variables, or it is substituted for one or more pieces of a variable's current value. Each variable is named by one glvn.

Each setargument is executed one at a time in left-to-right order. If the portion of the setargument to the left of the = does not consist of $X or $Y then the execution of a setargument occurs in the following order.

a) One of the following two operations is performed:

1) If the portion of the setargument to the left of the = consists of one or more glvns, the glvns are scanned in left-to-right order and all subscripts are evaluated, in left-to-right order within each glvn.

2) If the portion of the setargument to the left of the = consists of a setpiece or a setextract or a setqsub, the glvn that is the first argument of the setpiece or setextract or setqsub is scanned in left-to-right order and all subscripts are evaluated in left-to-right order within the glvn, and then the remaining arguments of the setpiece or setextract or setqsub are evaluated in left-to-right order.

b) The expr to the right of the = is evaluated. For each setleft, if it is a leftrestricted, the value to be assigned or replaced is truncated or converted to meet the inherent restrictions for that setleft before the assignment takes place. This means that in one SET statement, the various setlefts may receive different values.

c) One of the following five operations is performed.

1) If the left-hand side of the set is one or more glvns, the value of expr is given to each glvn, in left-to-right order. (See 7.1.2.2 for a description of the value assignment operation).

2) For each setleft that is a setpiece, of the form $PIECE(glvn,*d*,*m*,*n*), the value of expr replaces the *mth* through the *nth* pieces of the current value of the glvn, where the value of *d* is the piece delimiter. Note that both *m* and *n* are optional. If neither is present, then *m* = *n* = 1; if only *m* is present, then *n* = *m*. If glvn has no current value, the empty string is used as its current value. Note that the current value of glvn is obtained just prior to replacing it. That is, the other arguments of setpiece are evaluated in left-to-right order, and the expr to the right of the = is evaluated prior to obtaining the value of glvn.

Let *s* be the current value of glvn, *k* be the number of occurrences of *d* in *s*, that is, *k* = max(0,$LENGTH(*s*,*d*) − 1), and *t* be the value of expr. The following cases are defined, using the concatenation operator \_ of 7.2.1.1:

a) *m* > *n* or *n* < 1. The glvn is not changed and does not change the naked indicator.

b) *n* '< *m*−1 > *k*. The value in glvn is replaced by *s*\_F(*m*−1−*k*)\_*t*, where F(*x*) denotes a string of *x* occurrences of *d*, when *x* > 0; otherwise, F(*x*) = "". In either case, glvn affects the naked indicator.

c) *m*−1 '> *k* < *n*. The value in glvn is replaced by $P(*s*,*d*,1,*m*−1)\_F(min(*m*−1,1))\_*t*.

d) Otherwise, The value in glvn is replaced by $P(*s*,*d*,1,*m*−1)\_F(min(*m*−1,1))\_*t*\_*d*\_$P(*s*,*d*,*n*+1,*k*+1).

3) For each setleft that is a setextract of the form $EXTRACT(glvn,*m*,*n*), the value of expr replaces the *m*th through the *n*th characters of the current value of the glvn. Note that both *m* and *n* are optional. If neither is present, then *m* = *n* = 1; if only *m* is present, then *n* = *m*. If glvn has no current value, the empty string is used as its current value. Note that the current value of glvn is obtained just prior to replacing it. That is, the other arguments of setextract are evaluated in left-to-right order, and the expr to the right of the = is evaluated prior to obtaining the value of glvn.

Let *s* be the current value of glvn, *k* be the number of characters in *s*, that is, *k* = $LENGTH(*s*), and *t* be the value of expr. The following cases are defined, using the concatenation operator \_ of 7.2.1.1:

a) *m* > *n* or *n* < 1. The glvn is not changed and does not change the naked indicator.

b) *n* '< *m*-1 > *k*. The value in glvn is replaced by *s*\_$J("",*m*-1-*k*)\_*t*.

c) *m*-1 '> *k* < *n*. The value in glvn is replaced by $E(*s*,1,*m*-1)\_*t*.

d) Otherwise, The value in glvn is replaced by $E(*s*,1,*m*-1)\_*t*\_$E(*s*,*n*+1,*k*).

In cases b), c) and d) the naked indicator is affected.

4) If the left-hand side of the SET is a setev, one of the following two operations is performed:

a) If the setev is $ECODE, the current value of $ECODE is replaced by the value of expr. If the value of the expr is the empty string, $STACK($STACK,"ECODE") returns the empty string as do all forms of the function $STACK($STACK+*n*) for all values of *n* greater than 0. Note that if the value of $ECODE becomes non-empty, an error trap will be invoked.

b) If the setev is $ETRAP, the current value of $ETRAP is replaced by the value of expr.

5) For each setleft that is a setqsub of the form $QSUBSCRIPT(*nv*,*m*), if the value of *nv* is not a valid namevalue, an error condition occurs with ecode = “M90". Otherwise, let *t* be the value of expr and *nv* in the form NAME(*s1*,*s2*,...,*sn*), considering *n* to be zero if there are no subscripts, is modified according to the value of intexpr *m* as follows:

a) Values of *m* less than -1 are reserved for possible future use by the MDC.

b) If *m* = -1, the environment is changed to *t*.

c) If *m* = 0, the name is changed to *t*.

d) If *m* > *n*, the intervening *n* + 1 through *m* - 1 subscripts are each set to the empty string and the *mth* subscript is set to *t*.

e) Otherwise, the *mth* subscript is changed to *t*.

If the resulting value of *nv* is not a valid namevalue, an error condition occurs with ecode = “M90".

Note that the original and resulting namevalues are not “executed”, and will not modify the naked indicator beyond those modifications described at the end of this clause. Note also that the namevalues, while meeting the syntax of a namevalue, might specify a non-existent environment or contain a subscript value (such as the empty string or control characters) which do not meet the requirements of Section II Clause 2.3.3 (Values of subscripts).

If the portion of the setargument to the left of the = is a $X or a $Y then the execution of the setargument occurs in the following order:

a) The intexpr to the right of the = is evaluated.

b) The value of the intexpr is given to the special intrinsic variable on the left of the = with the following restrictions and affects:

1) The range of values of $X and $Y are defined in 7.1.4.10. Any attempt to set $X or $Y outside this range specified in 7.1.4.10 is erroneous (ecode="M43") and the value of $X or $Y will remain unchanged.

2) Setting $X or $Y changes the value of $X or $Y, respectively, but it does not cause any input or output operation. The purpose is to allow a program to correct the value of $X or $Y following input or output operations whose effect on the cursor position may not be reflected in $X and $Y.

The value of the naked indicator may be modified as a side-effect of the execution of a SET statement. Events that influence the value of the naked indicator are (in order of evaluation):

1) references to glvns in exprs in arguments or subscripts of setlefts;

2) references to glvns in the expr on the righthand side of the = sign;

3) references to glvns in the setdestination.

**8.2.22 TCOMMIT**

TC[OMMIT] postcond [ WS ]

If $TLEVEL is one, TCOMMIT performs a COMMIT of the TRANSACTION and sets $TRESTART to zero. (See the Transaction Processing subclause for the definition of COMMIT).

If $TLEVEL is greater than one, TCOMMIT subtracts one from $TLEVEL.

IF $TLEVEL is zero, TCOMMIT generates an error with ecode="M44".

Using the (model) linked list of RESTART CONTEXT-STRUCTUREs for the TRANSACTION, TCOMMIT removes the last created RESTART CONTEXT-STRUCTURE from both the PROCESS-STACK linked list and the TRANSACTION linked list and discards the RESTART CONTEXT-STRUCTURE.

**8.2.23 TRESTART**

TRE[START] postcond [ WS ]

If $TLEVEL is greater than zero, TRESTART performs a RESTART.

If $TLEVEL is zero, TRESTART generates an error with ecode="M44".

**8.2.24 TROLLBACK**

TRO[LLBACK] postcond [ WS ]

If $TLEVEL is greater than zero, a ROLLBACK is performed, $TLEVEL and $TRESTART are set to zero, and the naked indicator becomes undefined. (See the Transaction Processing subclause for the definition of ROLLBACK).

If $TLEVEL is zero, TROLLBACK generates an error with ecode="M44".

**8.2.25 TSTART**

TS[TART] postcond │ [ WS ] │

│ WS tstartargument │

tstartargument ::= │ [ restartargument ] [ : transparameters ] │

│ @ expratom V tstartargument │

╷ ╷

│ lname │

restartargument ::= │ ( L lname ) │

│ \* │

│ ( ) │

╵ ╵

╷ ╷

transparameters ::= │ tsparam │

│ ( tsparam [ : tsparam ] ... ) │

╵ ╵

tsparam ::= tstartkeyword [ = expr ]

│ S[ERIAL] │

tstartkeyword ::= │ T[RANSACTIONID] = expr │

│ Z[unspecified] [ = expr ] │

tstartkeywords that differ only in the use of corresponding upper and lower-case letters are equivalent.

Unused keywords other than those starting with the letter "Z" are reserved for future enhancement of the standard.

After evaluation of postcond, if any, and tstartargument, if any, TSTART adds one to $TLEVEL. If, as a result, $TLEVEL is one, then TSTART initiates a TRANSACTION that is restartable if a restartargument is present, or non-restartable if restartargument is absent; and serializable independently of LOCKs if transparameters are present and contain the keywords SERIAL or S, or dependent on LOCKs for serialization if those keywords are absent.

The tsparam, TRANSACTIONID, provides a means for identifying arbitrary classes of TRANSACTIONs.

The following discussion uses terms defined in the Variable Handling (see 7.1.2.2) and Process-Stack (see 7.1.2.3) models and, like those subclauses, does not imply a required implementation technique. TSTART creates a RESTART CONTEXT-STRUCTURE containing the execution location of the TSTART statement, the value for the naked indicator, a copy of the process LOCK-LIST, a RESTART NAME-TABLE and an exclusive indicator. TSTART attaches the CONTEXT-STRUCTURE to a linked list of such RESTART CONTEXT-STRUCTUREs for the current TRANSACTION and also to a linked list of CONTEXT-STRUCTUREs associated with the current PROCESS-STACK frame. TSTART copies from the currently active NAME-TABLE to the RESTART NAME-TABLE all entries corresponding to the local variable names specified by the restartargument. TSTART also points the entries in the RESTART NAME-TABLE to copies of VALUE-TABLE tuples containing values that persist unchanged from the point that the TSTART statement created the NAME-TABLE. When the restartargument is an asterisk (\*), it specifies all current names and causes the CONTEXT-STRUCTURE to be marked as exclusive.

**8.2.26 USE**

U[SE] postcond WS L useargument

╷ ┌╴ ┐ ╷

│ │ : deviceparameters │ │

│ devn │ │ │

useargument ::= │ │ : [ deviceparameters ] : mnemonicspace │ │

│ └╴ ┘ │

│ @ expratom V L useargument │

See 8.2.15 OPEN for mnemonicspace.

See 8.2.2 for the syntax and interpretation of devn and deviceparameters.

Before a device can be employed in conjunction with an input or output data transfer it must be designated, through execution of a USE statement, as the *current device*. Before a device can be named in an executed useargument, its ownership must have been established through execution of an OPEN statement.

The specified device remains current until such time as a new USE statement is executed. As a side effect of employing expr to designate a current device, $IO is given the value of expr contained in devn and $IOREFERENCE is given the value of devn.

Specification of device parameters, by means of the exprs in deviceparameters, is normally associated with the process of obtaining ownership; however, it is possible, by execution of a USE statement, to change the parameters of a device previously obtained.

Distinct values for $X and $Y are retained for each device. The special variables $X and $Y reflect those values for the current device. When the identity of the current device is changed as a result of the execution of a USE statement, the values of $X and $Y are saved, and the values associated with the new current device are then the values of $X and $Y.

**8.2.27 VIEW**

V[IEW] postcond arguments unspecified

VIEW makes available to the implementor a mechanism for examining machine-dependent information. It is to be understood that routines containing the VIEW statement may not be portable.

**8.2.28 WRITE**

W[RITE] postcond WS L writeargument

│ format │

writeargument ::= │ expr │

│ \* intexpr │

│ @ expratom V L writeargument │

The writearguments are executed, one at a time, in left-to-right order. Each form of argument defines an output operation to the current device.

When the form of argument is format, processing occurs in left-to-right order.

│ │

│ │ ! │ ... [ ? intexpr ] │

format ::= │ │ # │ │

│ │

│ ? intexpr │

│ │

│ /controlmnemonic [ ( expr [ , expr ] ... ) ] │

╵ ╵

╷ ╷ ┌─╴ ╶─┐

│ ? │ │ ident │

controlmnemonic ::= │ │ │ │ ...

│ ident │ │ digit │

╵ ╵ └─╴ ╶─┘

The following describes the effect of specific characters when used in a format:

! causes a *new line* operation on the current device. Its effect is the equivalent of writing CR LF on a pure ASCII device. In addition, $X is set to 0 and 1 is added to $Y.

# causes a *top of form* operation on the current device. Its effect is the equivalent of writing CR FF on a pure ASCII device. In addition, $X and $Y are set to 0. When the current device is a display, the screen is blanked and the cursor is positioned at the upper left-hand corner.

? intexpr

produces an effect similar to *tab to column intexpr*. If $X is greater than or equal to intexpr, there is no effect. Otherwise, the effect is the same as writing (intexpr − $X) spaces. (Note that the leftmost column of a line is column 0.)

/ controlmnemonic [ ( expr [ , expr ] ... ) ]

produces an effect which is defined by the mnemonicspace which has been assumed by default or has been selected in a previous mnemonicspace specification with a USE statement. The relevant control-function is indicated by means of the controlmnemonic which must be defined in the above-mentioned mnemonicspace. Possible parameters are given through the optional exprs. Controlmnemonics which start with the character "?" are implementor-specific.

The implementor may restrict the use of controlmnemonics in a device-dependant way. A reference to an undefined mnemonicspace or an undefined controlmnemonic is reflected in special variable $DEVICE.

When the form of argument is expr, the value of expr is sent to the device. The effect of this string at the device is defined by appropriate device handling.

When the form of the argument is \*intexpr, one character, not necessarily from the ASCII set and whose code is the number represented in decimal by the value of intexpr, is sent to the device. The effect of this character at the device may be defined by the implementor in a device-dependent manner.

As WRITE transmits characters one at a time, certain characters or character combinations represent device control functions, depending on the identity of the current device. To the extent that the supervisory function can detect these control characters or character sequences, they will alter $X and $Y as follows.

graphic : add 1 to $X

backspace : set $X = max($X−1,0)

line feed : add 1 to $Y

carriage return : set $X = 0

form feed : set $Y = 0, $X = 0

When a format specification is interpreted and the effect would cause the 'physical' external equivalent of $X and $Y to be modified, this effect will be reflected as far as possible in the values of the special variables $X and $Y.

Output operations, except when the form of the argument is \*intexpr, are affected by the Character Set Profile output-transform.

**8.2.29 XECUTE**

X[ECUTE] postcond WS L xargument

│ expr postcond │

xargument ::= │ │

│ @ expratom V L xargument │

XECUTE provides a means of executing Omega code that arises from the process of expression evaluation.

Each xargument is evaluated one at a time in left-to-right order. If the postcond in the xargument is present and its tvexpr is false, the xargument is not executed. Otherwise, if the value of expr is *x*, execution of the xargument is executed in a manner equivalent to execution of DO OB *x* CB.

**8.2.30 Z**

Z[unspecified] arguments unspecified

All statementwords in a given implementation which are not defined in the standard are to begin with the letter Z. This convention protects the standard for future enhancement.

8.3 Device Parameters

8.3.1 Output timeout

For any mnemonicspace the implementation may define a device parameter that causes an error condition when an output-producing argument of a READ or WRITE statement fails to complete execution within a specified time. If it is defined, the device parameter shall conform to this clause and to the related sections of 7.1.3.2.

This device parameter shall have the following form:

deviceparam ::= OUTTIMEOUT = numexpr

numexpr shall be interpreted as the value of a timeout (see 8.1.5). Should any subsequent output-producing argument of a READ or WRITE statement to the device fail to complete execution within that time, then

a) the OUTSTALLED member of ^$DEVICE, described in 7.1.3.2, shall assume the value 1, and

b) an error with ecode = “M100" shall occur.

Output timeout shall not apply to a device when

a) no OUTTIMEOUT deviceparam has executed for the device, or

b) the value of numexpr in the most recent OUTTIMEOUT is non-positive.

No more than one output timeout shall apply to one device at any time. That is, an OUTTIMEOUT deviceparam shall replace any pre-existing OUTTIMEOUT deviceparam.

Note: output timeout applies to the execution of READ or WRITE arguments, not to the delivery of data to a device.

9. Character Set Profile charset

A charset is a definition of the valid characters and their characteristics available to a process. The required characteristics for a fully defined charset are:

a) The character codes and their meaning

b) The definition of which character codes are valid in names

c) The available patcodes and their definitions

d) The collation order of character strings.

Note: a charset definition is not necessarily tied to any (natural) language and could be an arbitrary set of characters or a repertoire from another set, such as ISO 10646.

charset ::= descriptor

│ │

│ │ │ │ │ │

descriptor ::= │ │ │ [ descsep ] │ │ │ . . .

│ ident │ │ │ digit │ │

│ │

│ - │ *(note: hyphen)*

│ \_ │ *(note: underscore)*

│ % │

│ \* │

│ . │

descsep ::= │ / │

│ + │

│ : │

│ $ │

│ ! │

│ @ │

The definition of the contents of standardized charsets is in Annex A. Unused charset names beginning with the initial letter Y are available for usage by Omega programmers; those beginning with the initial letter Z are reserved for vendor-defined charsets; all other charset names are reserved for future enhancement of the standard.

**American National Standard for Information Systems - Programming Languages - Omega (Section 2: Omega Portability Requirements)**

**Introduction**

Section 2 highlights, for the benefit of implementors and application programmers, aspects of the language that must be accorded special attention if Omega program transferability (i.e., portability of source code between various Omega implementations) is to be achieved. It provides a specification of limits that must be observed by both implementors and programmers if portability is not to be ruled out. To this end, implementors must meet or exceed these limits, treating them as a minimum requirement. Any implementor who provides definitions in currently undefined areas must take into account that this action risks jeopardizing the upward compatibility of the implementation, upon subsequent revision of the Omega Language Specification. Application programmers striving to develop portable programs must take into account the danger of employing ``unilateral extensions'' to the language made available by the implementor.

The following definitions apply to the use of the terms *explicit limit* and *implicit limit* within this document. An explicit limit is one which applies directly to a referenced language construct. Implicit limits on language constructs are second-order effects resulting from explicit limits on other language constructs. For example, the explicit statement line length restriction places an implicit limit on the length of any construct which must be expressed entirely within a single statement line.

**1 Character Set**

The character set used for routines and data is restricted to the Character Set Profile M (as defined in Annex A).

**2 Expression elements**

**2.1 Names**

Portable name length is limited to thirty-one (31) characters. All characters in a name are significant in determining uniqueness. Therefore the length restriction places an implicit limit on the number of unique names on an implementation. If a name's length exceeds an implementor's limit an error condition occurs with ecode = "M56".

**2.2 External routines and names**

The externalroutinename namespace is unspecified, as this is a function of the binding, although at the present time, a maximum of twenty-four (24) characters allowed is placed upon externalroutinenames to be treated uniquely, although this should be viewed as a minimum number that needs to be handled rather than as the maximum number that can be used. Any number of characters, from one to the maximum number shall be valid as externalroutinenames. Any additional external mapping between these names and any actually used by an external package is an implementation issue.

**2.3 Local variables**

**2.3.1 Number of local variables**

The number of local variable names in existence at any time is not explicitly limited. However, there are implicit limitations due to the storage space restrictions (Clause 8).

**2.3.2 Number of subscripts**

There is no explicit limit on the number of distinct local variable nodes which may be defined, but there is an implicit limit based on the number of subscripts that may be defined for any local variable reference. The number of subscripts in a local variable is limited in that, in a local array reference, the total length of the array reference must not exceed 510. The length of an array reference, assuming it is in the form name ( i1 , i2 , ... , in ), is calculated as follows. If:

N = $L(name),

I = $L(i1) + $L(i2) + ... + $L(in), where each subscript (i1 through in) is either a numlit or a sublit, and

L = n,

then:

the total length of an array reference = N + I + ( 2 \* L ) + 15.

**2.3.3 Values of subscripts**

Local variable subscript values are nonempty strings which shall only contain characters from the M printable character subset. The length of individual subscripts is limited to 255 characters; in addition, a complete variable name reference is limited according to the restrictions specified in 2.3.2. When the subscript value satisfies the definition of a numeric data value (See 7.1.4.3 of Section 1), it is further subject to the restrictions of number range given in 2.6. The use of subscript values which do not meet these criteria is undefined, except for the use of the empty string as the last subscript of a starting reference in the context of data transversal functions such as $ORDER and $QUERY.

**2.4 Global variables**

**2.4.1 Number of global variables**

There is no explicit limit on the number of distinct global variable names in existence at any time.

**2.4.2 Number of subscripts**

The number of subscripts in a global variable is limited in that, in a global array reference, the total length of the array reference must not exceed 510. The length of an array reference, assuming it is in the form ^ VB environment VB name ( i1 , i2 , ... , in ), is calculated as follows. If:

E = $L(environment),

N = $L(name),

I = $L(i1) + $L(i2) + ... + $L(in), where each subscript (i1 through in) is either a numlit or a sublit, and

L = n,

then:

the total length of an array reference = E + 3 + N + I + ( 2 \* L ) + 15.

**2.4.3 Values of subscripts**

The restrictions imposed on the values of global variable subscripts are identical to those imposed on local variable subscripts (see 2.3.3).

**2.4.4 Number of nodes**

There is no explicit limit on the number of distinct global variable nodes which may be defined.

**2.5 Data types**

The Omega Language Specification defines a single data type, namely, variable length character strings. Contexts which demand a numeric, integer, or truth value interpretation are satisfied by unambiguous rules for mapping a string datum into a number, integer, or truth value.

The implementor is not limited to any particular internal representation. Any internal representation(s) may be employed as long as all necessary mode conversions are performed automatically and all external behavior agrees with the Omega Language Specification. For example, integers might be stored as binary integers and converted to decimal character strings whenever an operation requires a string value.

**2.6 Number range**

All values used in arithmetic operations or in any context requiring a numeric interpretation are within the inclusive intervals [-1025, -10-25] or [10-25, 1025], or are zero.

Implementations shall represent numeric quantities with at least 15 significant digits. The error introduced by any single instance of the arithmetic operations of addition, subtraction, multiplication, division, integer division, or modulo shall not exceed one part in 1015. The error introduced by exponentiation shall not exceed one part in 107.

Programmers should exercise caution in the use of noninteger arithmetic. In general, arithmetic operations on noninteger operands or arithmetic operations which produce noninteger results cannot be expected to be exact. In particular, noninteger arithmetic can yield unexpected results when used in loop control or arithmetic tests.

**2.7 Integers**

The magnitude of the value resulting from an integer interpretation is limited by the accuracy of numeric values (see 2.6). The values produced by integer valued operators and functions also fall within this range (see 7.1.4.6 of Section 1 for a precise definition of integer interpretation).

**2.8 Character strings**

Character string length is limited to 32,767 characters for local variables, 510 characters for global variables, and 32,767 characters for structured system variables. The characters permitted within character strings must include those defined in the ASCII Standard (ANSI X3.4-1986). If a string's length exceeds an implementor's limit, an error condition occurs with ecode="M75".

**2.9 Special variables**

The special variables $X and $Y are nonnegative integers (see 2.7). The effect of incrementing $X and $Y past the maximum allowable value is undefined. (For a description of the cases in which the values of $X and $Y may be altered see 8.2.27 of Section 1; for a description of the type of values $X and $Y may have see 7.1.4.10 of Section 1). The value of $SYSTEM as provided by an implementor must conform to the requirements for a local variable subscript (see 2.3.3).

**3 Expressions**

**3.1 Nesting of expressions**

The number of levels of nesting in expressions is not explicitly limited. The maximum string length does impose an implicit limit on this number (see 2.8).

**3.2 Results**

Any final result that does not satisfy the constraints on character strings (see 2.8) is erroneous. Any intermediate result that does not satisfy the constraints on local variable character strings (see 2.8) is erroneous. Furthermore, integer results are erroneous if they do not satisfy the constraints on integers (see 2.7).

**3.3 External References**

External references are not portable.

**4 Routines and statement lines**

**4.1 Statement lines**

A statement line (line) must satisfy the constraints on global variable character strings (see 2.8). The length of a statement line is the number of characters in the line up to but not including the eol.

The characters within a statement line are restricted to the 95 ASCII printable characters. The character set restriction places a corresponding implicit restriction upon the value of the argument of the indirection delimiter (Clause 7).

**4.2 Number of statement lines**

There is no explicit limit on the number of statement lines in a routine, subject to storage space restrictions (Clause 8).

**4.3 Number of statements**

The number of statements per line is limited only by the restriction on the maximum statement line length (see 4.1).

**4.4 Labels**

A label of the form name is subject to the constraint on names (see 2.1), with the exception that the first 31 characters are uniquely distinguished. Labels of the form intlit are subject to the same length constraints.

**4.5 Number of labels**

There is no explicit limit on the number of labels in a routine. However, the following restrictions apply:

a) A statement line may have only one label.

b) No two lines may be labeled with equivalent (not uniquely distinguishable) labels.

**4.6 Number of routines**

There is no explicit limit on the number of routines. The number of routines is implicitly limited by the name length restriction (see 2.1).

**5 External routine calls**

When the external routine called is not within the current default Omega environment, all variables should be assumed to be scalars (i.e., *a* refers to the value associated with *a*, but does not refer to any descendants *a* might have such as *a(1)*, etc.). No prohibition against non-scalar extensions should be inferred, only that they may not be portable. It should be noted that no all-encompassing implied guarantee of the number of routines supported by an external package exists.

**6 Character Set Profiles**

Character Set Profiles are registered through the MUMPS Development Committee (ANSI X11). New Character Set Profile Definitions are approved through the standard procedures of the MUMPS Development Committee.

Routines and data created using a registered Character Set Profile are portable to all implementations which support that Character Set Profile.

The list of MDC registered Character Set Profiles is included in Annex A.

Note that subscript-string length (see 2.3.2, 2.3.3, 2.4.2, 2.4.3) is either the length of the value of the subscript, or the length of the computed Character Set Profile collation value, whichever is larger.

Collation values are not portable between implementations unless the value is explicitly stated in the definition of the Character Set Profile.

**7 Indirection**

The values of the argument of indirection and the argument of the XECUTE statement are subject to the constraints on character string length (see 2.8). They are additionally restricted to the character set limitations of statement lines (see 4.1).

**8 Storage space restrictions**

The size of a single routine must not exceed 20,000 characters. The size of a routine is the sum of the sizes of all the lines in the routine. The size of each line is its length (as defined in 4.1) plus two.

The size of local variable storage must not exceed 100,000 characters. This size is defined as the sum of the sizes of all defined local variables, whether within the current NEW context or defined in a higher level NEW context. The size of an unsubscripted local variable is the length of its name in characters plus the length of its value in characters, plus four. The size of a local array is the sum of the following:

a) The length of the name of the array.

b) Four characters plus the length of each value.

c) The size of each subscript in each subscript list.

d) Two additional characters for each node *N*, whenever $DATA(*N*) is 10 or 11.

All subscripts and values are considered to be character strings for this purpose.

**9 Process-Stack**

Systems will provide a minimum of 127 levels in the PROCESS-STACK. The actual use of all these levels may be limited by storage restrictions (Clause 8).

Nesting within an expression is not counted in this limit. Expression nesting is not explicitly limited; however, it is implicitly limited by the storage restriction (Clause 8).

**10 Formats**

Device control may be effected through the READ and WRITE statements using the /controlmnemonic syntax in a specification of a format. In general, portability of routines containing such syntax is only possible in cases which meet several criteria, most obviously

a) the devices to be used at the receiving facility must have all the capabilities required by the /controlmnemonic occurrences in the routines;

b) the implementors of the systems at both the originating and the receiving facilities have implemented each combination of mnemonicspace and controlmnemonic in compatible ways.

As a result of these limitations, 'blind interchange' will only be dependent upon the devices at the receiving site.

However, the following advice to both implementors and programmers will increase the number of cases in which 'informed interchange' will be possible.

However user-defined mnemonicspaces, together with their associated controlmnemonics, are inherently portable provided that the Omega routines are also portable.

**10.1 mnemonicspace**

For portability, the mnemonicspace to be used must be a generally accepted standard, e.g. ANSI X3.64-199\_ or GKS, or after such a standard would have been accepted, any other ANSI or ISO standard.

**10.2 controlmnemonic**

For portability, the controlmnemonic must be one of the controlmnemonics assigned to a control-function specified in the chosen mnemonicspace and interpretation of the format specification must lead to the effect described in the mnemonicspace. There should be no other (side-)effects on the device.

With regard to the status of the process, the value of some special variables may change, e.g. with some control-functions $X and $Y would have to receive proper values. Apart from these documented effects, no other effects may be caused by any implementation.

An implementation needs not to allow for all controlmnemonics in all mnemonicspaces.

**10.3 Parameters**

A format containing /controlmnemonic may contain one or more parameters, specified as L expr, in which case each expr specifies a parameter of the control-function. The exprs must appear in the same order and number as the parameters in the corresponding mnemonicspace. The value of each expr should meet the limitations of 2.6 through 2.8.

**11 Transaction processing**

**11.1 Number of modifications in a TRANSACTION**

The sum of the lengths of the namevalues and values of global variable tuples modified within a TRANSACTION must not exceed 57,343 characters.

**11.2 Number of nested TSTARTs within a TRANSACTION**

A single TRANSACTION must not contain more than 126 TSTARTs after the TSTART that initiates the TRANSACTION.

**12 Other portability requirements**

Programmers should exercise caution in the use of noninteger values for the HANG statement and in timeouts. In general, the period of actual time which elapses upon the execution of a HANG statement cannot be expected to be exact. In particular, relying upon noninteger values in these situations can lead to unexpected results.

Implementations may restrict access to ssvns that contain default environments of processes other than the one referring to the ssvn. Therefore, portable programs shall not rely on the ssvns defined in 7.1.3.10 when processid is not their own $JOB.

**American National Standard for Information Systems - Programming Languages - Omega (Section 3: X3.64 Binding)**

**Introduction**

ANSI X3.64 is a functional standard for additional control functions for data interchange with two-dimensional character-imaging input and/or output devices. It is an ANSI standard, but also an ISO standard with roughly similar characteristics exists (ISO 2022). As such, it has been implemented in many devices worldwide. It is expected that Omega can be easily adapted to these implementations.

The standard defined as ANSI X3.64 defines a format for device-control. No physical device is required to be able to perform all possible control-functions. In reality, as some functions rely on certain physical properties of specific devices, no device will be able to perform all functions. The standard, however, does not specify which functions a device should be able to do, but if it is able to perform a function, how the control-information for this function is to be specified.

This binding is to the functional definitions included in X3.64. The actual dialogue between the Omega implementation and the device is left to the implementor.

**1 The binding**

ANSI X3.64 is accessed from the Omega language by making use of mnemonicspaces. A controlmnemonic from X3.64 may be accessed as follows:

/controlmnemonic [ ( expr [, expr ] ... ) ]

where the relevant controlmnemonic equalling the generic function and exprs the possible applicable parameters. The use of a controlmnemonic produces the effect defined in ANSI X3.64 for the control-function with the same name as the controlmnemonic specified.

Some controlmnemonics return a value, or a collection of values. It is perfectly legal to issue these controlmnemonics with either a READ or WRITE statement. If a READ statement is used, the argument list in the statement(s) must be ordered to correctly accept the returned values. If a WRITE statement is used the values returned may be read by a single, or series of, READ statements. These READ statements must be correctly ordered to match the returned values, however there may be intermediate calculations utilizing some of the returned values before reading the remaining values in the list. Reading the return list of values may be terminated without error by issuing another controlmnemonic. In this case, all returned values not assigned to a variable will be lost to the application program.

All controlmnemonics have the same name in Omega as in X3.64.

Unless explicitly mentioned, the use of X3.64 controlmnemonics has no side-effects on special variables such as $X, $Y, $KEY and $DEVICE.

**1.1 Control-functions with an effect on $X or $Y or both**

Below follows a list of control-functions (X3.64) or controlmnemonics (M) that have an effect on the special variables $X or $Y or both. Since some definitions in X3.64 are fairly open-ended, the exact effect may be implementation dependent in some cases. In section 3.4 these open-ended definitions are listed resolution of possible ambiguities are stated.

The relevant controlmnemonics are:

/CBT(n) $X

/CHA(x) $X

/CHT(n) $X

/CNL(n) $X, $Y

/CPL(n) $X, $Y

/CUB(n) $X

/CUD(n) $Y

/CUF(n) $X

/CUP(y,x) $X, $Y

/CUU(n) $Y

/CVT(n) $Y

/HPA(x) $X

/HPR(n) $X

/HTJ $X

/HVP(y,x) $X, $Y

/IND $Y

/NEL $X, $Y

/PLD $Y

/PLU $Y

/REP(n) $X, $Y

/RI $Y

/RIS $X=0, $Y=0

/VPA(y) $Y

/VPR(n) $Y

The control-function REP repeats the previous character or function as many times as indicated by its argument. Hence, the side-effects of this function do not depend on this function itself, but rather on the character or function that is being repeated.

**1.2 Control-functions with an effect on $KEY**

Currently only one controlmnemonic may have a side-effect on special variable $KEY: /DSR (device status report). The side-effect depends on the value of the parameter of this function: parameter-value 0 or 5 will cause a status report to be returned, parameter-value 6 will cause the active cursor-position to be returned. The format of the value returned is:

$CHAR(27,91)\_REPORT\_$CHAR(110)

or

$CHAR(27,91)\_Y\_$CHAR(59)\_X\_$CHAR(82)

where REPORT is a code for the status reported, Y is the value of the current Y-coordinate and X is the value of the current X-coordinate.

The values described will be reported in special variable $KEY as a side-effect of the first READ statement that is executed after the control-function has been issued.

**1.3 Control-functions with an effect on $DEVICE**

All controlmnemonics will have a side-effect on special variable $DEVICE. The most common situation will be that $DEVICE will receive the value:

"0,,X3-64"

in order to reflect the correct processing of a controlmnemonic.

In certain situations a status has to be indicated. Status codes for $DEVICE relating to X3.64 are as follows:

code American English Description

1 mnemonicspace not found

2 invalid mnemonic

3 parameter out of range

4 hardware error

5 mnemonic not available for this device

6 parameter not available for this device

7 attempt to move outside boundary - not moved

8 attempt to move outside boundary - moved to boundary

9 auxiliary device not ready

**1.4 Open-ended definitions**

Under some conditions, the behavior specified by X3.64 is either ambiguous or optional. The following clarifies the behavior to ensure consistency:

CBT Move the cursor to the last horizontal tabulator-stop in the previous line. If no such tabulator-stop exists, don't move the cursor.

CHA when a location outside the available horizontal range is specified:

Move the cursor in the direction suggested by the parameter-value to either the rightmost (parameter value greater than current position) or leftmost (parameter value less than current position) position.

CHT when no further forward horizontal tabulator-stops have been defined in the current line:

Move the cursor to the first horizontal tabulator-stop in the next line. If no such tabulator-stop exists, don't move the cursor.

CNL when the cursor is moved forward beyond the last line on the device:

Do not move the cursor. If the output device is a CRT-screen, scroll up one line.

CPL when the cursor is moved backward beyond the first line on the device:

Do not move the cursor. If the output device is a CRT-screen, scroll down one line.

CUB when the cursor is moved backward beyond the first position on a line:

Do not move the cursor.

CUD when the cursor is moved downward beyond the last line on a device:

Do not move the cursor.

CUF when the cursor is moved forward beyond the last position on a line:

Do not move the cursor.

CUP when a location outside the available horizontal or vertical ranges is specified:

Do not move the cursor.

CUU when the cursor is moved upward beyond the last line on a device:

Do not move the cursor.

CVT when no further forward vertical tabulator-stops have been defined on the device:

Move the cursor to the first vertical tabulator-stop in the next page. If no such tabulator-stop exists, don't move the cursor.

HPA when a location outside the available horizontal range is specified:

Move the cursor in the direction suggested by the parameter-value to either the rightmost (parameter value greater than current position) or leftmost (parameter value less than current position) position.

HPR when a location outside the available horizontal range is specified:

Move the cursor in the direction suggested by the parameter-value to either the rightmost (parameter value positive) or leftmost (parameter value negative) position.

HTJ when no further forward horizontal tabulator-stops have been defined in the current line:

Move the cursor to the first horizontal tabulator-stop in the next line. If no such tabulator-stop exists, don't move the cursor.

HVP when a location outside the available horizontal or vertical ranges is specified:

Do not move the cursor.

IND when the cursor is moved downward beyond the last line on a device:

Move the cursor to the corresponding horizontal position in the first line on the next page.

NEL when the cursor is moved downward beyond the last line on a device:

Move the cursor to the first position on the first line on the next page.

PLD this function may or may not be similar to CUD or IND. The effect of two successive PLD operations may or may not be equal to the effect of one single CUD or IND operation:

This function will be identical to CUD.

The effect of PLD and PLU will be complementary, i.e. .PLD immediately followed by PLU will effectively not move the cursor.

PLU this function may or may not be similar to CUU or RI. The effect of two successive PLU operations may or may not be equal to the effect of one single CUU or RI operation:

This function will be identical to CUU.

The effect of PLD and PLU will be complementary, i.e. .PLU immediately followed by PLD will effectively not move the cursor.

RI when the cursor is moved upward beyond the first line on a device:

Move the cursor to the corresponding horizontal position in the last line on the previous page.

VPA when a location outside the vertical range is specified:

Move the cursor in the direction suggested by the parameter-value to either the bottommost (parameter value greater than current position) or topmost (parameter value less than current position) position.

VPR when a location outside the vertical range is specified:

Move the cursor in the direction suggested by the parameter-value to either the bottommost (parameter value positive) or topmost (parameter value negative) position.

The following functions shall not cause the cursor to move: ICH, JFY, MC, NP, DL and PP.

The following functions shall move the cursor so that it will point to the same character in the new projection of the information: SD, SL, SR and SU. Boundary conditions will be similar to CUD, CUB, CUF and CUU respectively.

**2 Portability issues**

**2.1 Implementation**

Any implementation of this binding will accept all controlmnemonics specified. However, in most cases all controlmnemonics will not be supported for all devices. The appropriate error code will be return in $DEVICE to indicate if a particular controlmnemonic is supported for the current device.

**2.2 Application**

Several controlmnemonics specified in X3.64 are ambiguous and usage of these will likely have different meaning between different devices and implementations. Usage of these will not be portable.

Control-

mnemonic Control Function

APC Application Program Statement

DA Device Attributes

DCS Device Control String

FNT Font Selection

INT Interrupt

OSC Operating System Statement

PLD Partial Line Down (CUD recommended; see 1.4)

PLU Partial Line Up (CUU recommended; see 1.4)

PM Privacy Message

PU1 Private Use One

PU2 Private Use Two

SGR Select Graphic Rendition for the following:

10 primary font

11 first alternative font

12 second alternative font

13 third alternative font

14 forth alternative font

15 fifth alternative font

16 sixth alternative font

17 seventh alternative font

18 eighth alternative font

19 ninth alternative font

SS2 Single Shift Two

SS3 Single Shift Three

**3 Conformance**

Each implementation must supply a list of the controlmnemonic and arguments that are supported for each device.

**Annex A: Character Set Profiles** (normative)

The definition of a Character Set Profile requires the definition of four elements ── the names of the characters in the character set and the internal codes which are used to represent them, the definitions of which characters match which pattern codes, the collation scheme used, and the definition of which characters may be used in names.

Note that the patcodes A, C, E, L, N, P, and U are applicable for all character set profiles; in addition patcode E matches any character, not just those listed in any specific charset.

Two collation schemes are provided which only require a properly defined table of characters for the Character Set associated with the specific Character Set Profile.

*STRING* COLLATION

Determining the Collation Ordering for a Character Set Profile requires the collation value(s) for each character within the character set be accessible a group of values presented as an *n*-tuple. Each column of the definition table provides one value of the tuple in the specified order. When no value is present in any colum, the corresponding character id value is used in its place. Note that certain characters may be represented with more than one value entry line in the table; in these cases the entries are taken one at a time and treated as if they represented separate characters in the original string (e.g., the character Æ in ISO-Latin-1 (id# 198) would be treated as a form of the string "AE").

Let *s* be any non-empty string. Define the numeric function CVn(*s*) to return the nth-order collation value for string *s*: unless otherwise specified this value is determined by evaluating the value in the nth column of each collation tuple for each character in the string examined in left-to-right order and combining them together. Note: selected collation-tuple columns may optionally be designated for right-to-left evaluation.

The Collation Ordering function CO determines relative ordering for a character set. The exact value of this function is not specified here, however, the values formed by any implementation must satisfy the following ruls when comparing two non-equal strings:

Let *t* also be any non-empty string, not equal to *s*. The *STRING* Collation Ordering function CO is defined as:

a) CO( "" , *s* ) = *s*

b) CO( *s* , *t* ) = *t* if, and only if, CVj(*t*) > CVj(*s*)

and for all I, I=1 ... j-1, CVi(*t*) = CVi(*s*);

otherwise CO(*s*,*t*) = *s*.

*M* COLLATION

The *M* Collation Ordering function CO uses the definition of CVn(*s*) specified in *STRING* Collation and is otherwise different only with respect to numbers:

Let *s* be any non-empty string, let *m* and *n* be strings satisfying the definition of numeric data values (see I.7.1.4.3), and *u* and *v* be non-empty strings which do not satisfy that definition.

a) CO( "" , *s* ) = *s*

b) CO( *m* , *n* ) = *n* if *n* > *m*; otherwise, CO( *m* , *n* ) = *m*

c) CO( *m* , *u* ) = *u*

d) CO( *u* , *v* ) = *v* if, and only if, CVj(*v*) > CVj(*u*)

and for all I, I=1 ... j-1, CVi(*v*) = CVi(*u*);

otherwise, CO(*u*,*v*)=*u*.

**1 charset M**

The charset M is defined using the table A.1. The values in the columns headed Character ID and Character Symbol are taken from ASCII (X3.4-1990). The column headed patcode defines which characters match the patcodes A, C, E, L, N, P, and U. The characters in the table with a patcode of A are defined as idents. The collation rule used is *M* collation, using the collation order values provided in the table.

**2 charset ASCII**

The charset ASCII is defined using the table A.1. The values in the columns headed Character ID and Character Symbol are taken from ASCII (X3.4-1990). The column headed patcode defines which characters match the patcodes A, C, E, L, N, P, and U. The characters in the table with a patcode of A are defined as idents. The collation rule used if *STRING* collation, using the collation order values provided in the table.

**Table A.1 - ASCII Character Set Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Character  ID | Character Symbol | patcode | Collation Table | | |
| 1st Order | 2nd Order | 3rd Order |
| 0 | *NUL* | C,E | 0 |  |  |
| 1 | *SOH* | C,E | 1 |  |  |
| 2 | *STX* | C,E | 2 |  |  |
| 3 | *ETX* | C,E | 3 |  |  |
| 4 | *EOT* | C,E | 4 |  |  |
| 5 | *ENQ* | C,E | 5 |  |  |
| 6 | *ACK* | C,E | 6 |  |  |
| 7 | *BELL* | C,E | 7 |  |  |
| 8 | *BS* | C,E | 8 |  |  |
| 9 | *HT* | C,E | 9 |  |  |
| 10 | *LF* | C,E | 10 |  |  |
| 11 | *VT* | C,E | 11 |  |  |
| 12 | *FF* | C,E | 12 |  |  |
| 13 | *CR* | C,E | 13 |  |  |
| 14 | *SO* | C,E | 14 |  |  |
| 15 | *SI* | C,E | 15 |  |  |
| 16 | *DLE* | C,E | 16 |  |  |
| 17 | *DC1* | C,E | 17 |  |  |
| 18 | *DC2* | C,E | 18 |  |  |
| 19 | *DC3* | C,E | 19 |  |  |
| 20 | *DC4* | C,E | 20 |  |  |
| 21 | *NAK* | C,E | 21 |  |  |
| 22 | *SYN* | C,E | 22 |  |  |
| 23 | *ETB* | C,E | 23 |  |  |
| 24 | *CAN* | C,E | 24 |  |  |
| 25 | *EM* | C,E | 25 |  |  |
| 26 | *SUB* | C,E | 26 |  |  |
| 27 | *ESC* | C,E | 27 |  |  |
| 28 | *FS* | C,E | 28 |  |  |
| 29 | *GS* | C,E | 29 |  |  |
| 30 | *RS* | C,E | 30 |  |  |
| 31 | *US* | C,E | 31 |  |  |
| 32 | SP (space) | P,E | 32 |  |  |
| 33 | ! | P,E | 33 |  |  |
| 34 | " | P,E | 34 |  |  |
| 35 | # | P,E | 35 |  |  |
| 36 | $ | P,E | 36 |  |  |
| 37 | % | P,E | 37 |  |  |
| 38 | & | P,E | 38 |  |  |
| 39 | ' (apostrophe) | P,E | 39 |  |  |
| 40 | ( | P,E | 40 |  |  |
| 41 | ) | P,E | 41 |  |  |
| 42 | \* | P,E | 42 |  |  |
| 43 | + | P,E | 43 |  |  |
| 44 | , (comma) | P,E | 44 |  |  |
| 45 | - (hyphen) | P,E | 45 |  |  |
| 46 | . | P,E | 46 |  |  |
| 47 | / | P,E | 47 |  |  |
| 48 | 0 | N,E | 48 |  |  |
| 49 | 1 | N,E | 49 |  |  |
| 50 | 2 | N,E | 50 |  |  |
| 51 | 3 | N,E | 51 |  |  |
| 52 | 4 | N,E | 52 |  |  |
| 53 | 5 | N,E | 53 |  |  |
| 54 | 6 | N,E | 54 |  |  |
| 55 | 7 | N,E | 55 |  |  |
| 56 | 8 | N,E | 56 |  |  |
| 57 | 9 | N,E | 57 |  |  |
| 58 | : | P,E | 58 |  |  |
| 59 | ; | P,E | 59 |  |  |
| 60 | < | P,E | 60 |  |  |
| 61 | = | P,E | 61 |  |  |
| 62 | > | P,E | 62 |  |  |
| 63 | ? | P,E | 63 |  |  |
| 64 | @ | P,E | 64 |  |  |
| 65 | A | A,U,E | 65 |  |  |
| 66 | B | A,U,E | 66 |  |  |
| 67 | C | A,U,E | 67 |  |  |
| 68 | D | A,U,E | 68 |  |  |
| 69 | E | A,U,E | 69 |  |  |
| 70 | F | A,U,E | 70 |  |  |
| 71 | G | A,U,E | 71 |  |  |
| 72 | H | A,U,E | 72 |  |  |
| 73 | I | A,U,E | 73 |  |  |
| 74 | J | A,U,E | 74 |  |  |
| 75 | K | A,U,E | 75 |  |  |
| 76 | L | A,U,E | 76 |  |  |
| 77 | M | A,U,E | 77 |  |  |
| 78 | N | A,U,E | 78 |  |  |
| 79 | O | A,U,E | 79 |  |  |
| 80 | P | A,U,E | 80 |  |  |
| 81 | Q | A,U,E | 81 |  |  |
| 82 | R | A,U,E | 82 |  |  |
| 83 | S | A,U,E | 83 |  |  |
| 84 | T | A,U,E | 84 |  |  |
| 85 | U | A,U,E | 85 |  |  |
| 86 | V | A,U,E | 86 |  |  |
| 87 | W | A,U,E | 87 |  |  |
| 88 | X | A,U,E | 88 |  |  |
| 89 | Y | A,U,E | 89 |  |  |
| 90 | Z | A,U,E | 90 |  |  |
| 91 | [ | P,E | 91 |  |  |
| 92 | \ | P,E | 92 |  |  |
| 93 | ] | P,E | 93 |  |  |
| 94 | ^ | P,E | 94 |  |  |
| 95 | \_ (underscore) | P,E | 95 |  |  |
| 96 | ` | P,E | 96 |  |  |
| 97 | a | A,L,E | 97 |  |  |
| 98 | b | A,L,E | 98 |  |  |
| 99 | c | A,L,E | 99 |  |  |
| 100 | d | A,L,E | 100 |  |  |
| 101 | e | A,L,E | 101 |  |  |
| 102 | f | A,L,E | 102 |  |  |
| 103 | g | A,L,E | 103 |  |  |
| 104 | h | A,L,E | 104 |  |  |
| 105 | i | A,L,E | 105 |  |  |
| 106 | j | A,L,E | 106 |  |  |
| 107 | k | A,L,E | 107 |  |  |
| 108 | l | A,L,E | 108 |  |  |
| 109 | m | A,L,E | 109 |  |  |
| 110 | n | A,L,E | 110 |  |  |
| 111 | o | A,L,E | 111 |  |  |
| 112 | p | A,L,E | 112 |  |  |
| 113 | q | A,L,E | 113 |  |  |
| 114 | r | A,L,E | 114 |  |  |
| 115 | s | A,L,E | 115 |  |  |
| 116 | t | A,L,E | 116 |  |  |
| 117 | u | A,L,E | 117 |  |  |
| 118 | v | A,L,E | 118 |  |  |
| 119 | w | A,L,E | 119 |  |  |
| 120 | x | A,L,E | 120 |  |  |
| 121 | y | A,L,E | 121 |  |  |
| 122 | z | A,L,E | 122 |  |  |
| 123 | { | P,E | 123 |  |  |
| 124 | | | P,E | 124 |  |  |
| 125 | } | P,E | 125 |  |  |
| 126 | ~ | P,E | 126 |  |  |
| 127 | *DEL* | C,E | 127 |  |  |

Note: 2nd and 3rd order collation values happen to be blank (i.e., not needed) for this Character Set Profile definition; the 1st order collation value happens to be unique across all the characters in this profile.

**3 charset JIS90**

The charset JIS90 supports an encoding of Japanese characters. The specification for this was developed by the MUMPS Development Coordinating Committee - Japan and is described in JIS X0201-1990 and JIS X0208-1990. The English translation is partially reproduced in Annex G for information purposes. The reader should refer to JIS X0201-1990 and JIS X0208-1990 for full definition.

(Note that Annex G is informational.)

**4 charset ISO-8859-USA**

The charset ISO-8859-1-USA is defined using the table A.2. The values in the columns headed Character ID and Character Symbol are taken from ISO-8859-1 (ISO Latin 1). The column headed patcode defines which characters match the patcodes A, C, E, L, N, P, and U. The characters in the table with a patcode of A are defined as idents. The collation rule used is *STRING* collation, using the collation order values provided in the table: note that all collation is left-to-right precedence. Note also that the patcode I matches any non-ASCII characters (id# greater than 127), not just those listed in this charset.

**5 charset ISO-8859-1-USA/M**

The charset ISO-8859-1-USA/M is defined using the table A.2. The values in the columns headed Character ID and Character Symbol are taken from ISO-8859-1 (ISO Latin 1). The column headed patcode defines which characters match the patcodes A, C, E, L, N, P, and U. The characters in the table with a patcode of A are defined as idents. The collation rule used is *M* collation, using the collation order values provided in the table: note that all collation is left-to-right precedence. Note also that the patcode I matches any non-ASCII characters (id# greater than 127), not just those listed in this charset.

**Table A.2 - ISO-8859-1-USA Character Set Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Character  ID | Character Symbol | patcode | Collation Table | | |
| 1st Order | 2nd Order | 3rd Order |
| 0 | *NUL* | C,E | 0 |  |  |
| 1 | *SOH* | C,E | 1 |  |  |
| 2 | *STX* | C,E | 2 |  |  |
| 3 | *ETX* | C,E | 3 |  |  |
| 4 | *EOT* | C,E | 4 |  |  |
| 5 | *ENQ* | C,E | 5 |  |  |
| 6 | *ACK* | C,E | 6 |  |  |
| 7 | *BELL* | C,E | 7 |  |  |
| 8 | *BS* | C,E | 8 |  |  |
| 9 | *HT* | C,E | 9 |  |  |
| 10 | *LF* | C,E | 10 |  |  |
| 11 | *VT* | C,E | 11 |  |  |
| 12 | *FF* | C,E | 12 |  |  |
| 13 | *CR* | C,E | 13 |  |  |
| 14 | *SO* | C,E | 14 |  |  |
| 15 | *SI* | C,E | 15 |  |  |
| 16 | *DLE* | C,E | 16 |  |  |
| 17 | *DC1* | C,E | 17 |  |  |
| 18 | *DC2* | C,E | 18 |  |  |
| 19 | *DC3* | C,E | 19 |  |  |
| 20 | *DC4* | C,E | 20 |  |  |
| 21 | *NAK* | C,E | 21 |  |  |
| 22 | *SYN* | C,E | 22 |  |  |
| 23 | *ETB* | C,E | 23 |  |  |
| 24 | *CAN* | C,E | 24 |  |  |
| 25 | *EM* | C,E | 25 |  |  |
| 26 | *SUB* | C,E | 26 |  |  |
| 27 | *ESC* | C,E | 27 |  |  |
| 28 | *FS* | C,E | 28 |  |  |
| 29 | *GS* | C,E | 29 |  |  |
| 30 | *RS* | C,E | 30 |  |  |
| 31 | *US* | C,E | 31 |  |  |
| 32 | SP (space) | P,E | 32 |  |  |
| 33 | ! | P,E | 33 |  |  |
| 34 | " | P,E | 34 |  |  |
| 35 | # | P,E | 35 |  |  |
| 36 | $ | P,E | 36 |  |  |
| 37 | % | P,E | 37 |  |  |
| 38 | & | P,E | 38 |  |  |
| 39 | ' (apostrophe) | P,E | 39 |  |  |
| 40 | ( | P,E | 40 |  |  |
| 41 | ) | P,E | 41 |  |  |
| 42 | \* | P,E | 42 |  |  |
| 43 | + | P,E | 43 |  |  |
| 44 | , (comma) | P,E | 44 |  |  |
| 45 | - (hyphen) | P,E | 45 |  |  |
| 46 | . | P,E | 46 |  |  |
| 47 | / | P,E | 47 |  |  |
| 48 | 0 | N,E | 48 |  |  |
| 49 | 1 | N,E | 49 |  |  |
| 50 | 2 | N,E | 50 |  |  |
| 51 | 3 | N,E | 51 |  |  |
| 52 | 4 | N,E | 52 |  |  |
| 53 | 5 | N,E | 53 |  |  |
| 54 | 6 | N,E | 54 |  |  |
| 55 | 7 | N,E | 55 |  |  |
| 56 | 8 | N,E | 56 |  |  |
| 57 | 9 | N,E | 57 |  |  |
| 58 | : | P,E | 58 |  |  |
| 59 | ; | P,E | 59 |  |  |
| 60 | < | P,E | 60 |  |  |
| 61 | = | P,E | 61 |  |  |
| 62 | > | P,E | 62 |  |  |
| 63 | ? | P,E | 63 |  |  |
| 64 | @ | P,E | 64 |  |  |
| 65 | A | A,U,E | 65 | 1 | 1 |
| 66 | B | A,U,E | 66 | 1 | 1 |
| 67 | C | A,U,E | 67 | 1 | 1 |
| 68 | D | A,U,E | 68 | 1 | 1 |
| 69 | E | A,U,E | 70 | 1 | 1 |
| 70 | F | A,U,E | 71 | 1 | 1 |
| 71 | G | A,U,E | 72 | 1 | 1 |
| 72 | H | A,U,E | 73 | 1 | 1 |
| 73 | I | A,U,E | 74 | 1 | 1 |
| 74 | J | A,U,E | 75 | 1 | 1 |
| 75 | K | A,U,E | 76 | 1 | 1 |
| 76 | L | A,U,E | 77 | 1 | 1 |
| 77 | M | A,U,E | 78 | 1 | 1 |
| 78 | N | A,U,E | 79 | 1 | 1 |
| 79 | O | A,U,E | 80 | 1 | 1 |
| 80 | P | A,U,E | 81 | 1 | 1 |
| 81 | Q | A,U,E | 82 | 1 | 1 |
| 82 | R | A,U,E | 83 | 1 | 1 |
| 83 | S | A,U,E | 84 | 1 | 1 |
| 84 | T | A,U,E | 85 | 1 | 1 |
| 85 | U | A,U,E | 86 | 1 | 1 |
| 86 | V | A,U,E | 87 | 1 | 1 |
| 87 | W | A,U,E | 88 | 1 | 1 |
| 88 | X | A,U,E | 89 | 1 | 1 |
| 89 | Y | A,U,E | 90 | 1 | 1 |
| 90 | Z | A,U,E | 91 | 1 | 1 |
| 91 | [ | P,E | 93 |  |  |
| 92 | \ | P,E | 94 |  |  |
| 93 | ] | P,E | 95 |  |  |
| 94 | ^ | P,E | 96 |  |  |
| 95 | \_ (underscore) | P,E | 97 |  |  |
| 96 | ` | P,E | 98 |  |  |
| 97 | a | A,L,E | 65 | 0 | 1 |
| 98 | b | A,L,E | 66 | 0 | 1 |
| 99 | c | A,L,E | 67 | 0 | 1 |
| 100 | d | A,L,E | 68 | 0 | 1 |
| 101 | e | A,L,E | 70 | 0 | 1 |
| 102 | f | A,L,E | 71 | 0 | 1 |
| 103 | g | A,L,E | 72 | 0 | 1 |
| 104 | h | A,L,E | 73 | 0 | 1 |
| 105 | i | A,L,E | 74 | 0 | 1 |
| 106 | j | A,L,E | 75 | 0 | 1 |
| 107 | k | A,L,E | 76 | 0 | 1 |
| 108 | l | A,L,E | 77 | 0 | 1 |
| 109 | m | A,L,E | 78 | 0 | 1 |
| 110 | n | A,L,E | 79 | 0 | 1 |
| 111 | o | A,L,E | 80 | 0 | 1 |
| 112 | p | A,L,E | 81 | 0 | 1 |
| 113 | q | A,L,E | 82 | 0 | 1 |
| 114 | r | A,L,E | 83 | 0 | 1 |
| 115 | s | A,L,E | 84 | 0 | 1 |
| 116 | t | A,L,E | 85 | 0 | 1 |
| 117 | u | A,L,E | 86 | 0 | 1 |
| 118 | v | A,L,E | 87 | 0 | 1 |
| 119 | w | A,L,E | 88 | 0 | 1 |
| 120 | x | A,L,E | 89 | 0 | 1 |
| 121 | y | A,L,E | 90 | 0 | 1 |
| 122 | z | A,L,E | 91 | 0 | 1 |
| 123 | { | P,E | 99 |  |  |
| 124 | | | P,E | 100 |  |  |
| 125 | } | P,E | 101 |  |  |
| 126 | ~ | P,E | 102 |  |  |
| 127 | *DEL* | C,E | 103 |  |  |
| 128 |  | C,E,I | 104 |  |  |
| 129 |  | C,E,I | 105 |  |  |
| 130 |  | C,E,I | 106 |  |  |
| 131 |  | C,E,I | 107 |  |  |
| 132 | *IND* | C,E,I | 108 |  |  |
| 133 | *NEL* | C,E,I | 109 |  |  |
| 134 | *SSA* | C,E,I | 110 |  |  |
| 135 | *HTS* | C,E,I | 111 |  |  |
| 136 | *HTJ* | C,E,I | 112 |  |  |
| 137 | *VTS* | C,E,I | 113 |  |  |
| 138 | *PLD* | C,E,I | 114 |  |  |
| 139 | *PLU* | C,E,I | 115 |  |  |
| 140 | *RI* | C,E,I | 116 |  |  |
| 141 | *SS2* | C,E,I | 117 |  |  |
| 142 | *SS3* | C,E,I | 118 |  |  |
| 143 | *DCS* | C,E,I | 119 |  |  |
| 144 | *PU1* | C,E,I | 120 |  |  |
| 145 | *PU2* | C,E,I | 121 |  |  |
| 146 | *STS* | C,E,I | 122 |  |  |
| 147 | *CCH* | C,E,I | 123 |  |  |
| 148 | *MW* | C,E,I | 124 |  |  |
| 149 | *SPA* | C,E,I | 125 |  |  |
| 150 | *EPA* | C,E,I | 126 |  |  |
| 151 |  | C,E,I | 127 |  |  |
| 152 |  | C,E,I | 128 |  |  |
| 153 |  | C,E,I | 129 |  |  |
| 154 |  | C,E,I | 130 |  |  |
| 155 | *CSI* | C,E,I | 131 |  |  |
| 156 | *ST* | C,E,I | 132 |  |  |
| 157 | *OSC* | C,E,I | 133 |  |  |
| 158 | *PM* | C,E,I | 134 |  |  |
| 159 | *APC* | C,E,I | 135 |  |  |
| 160 | *NBSP* | C,E,I | 136 |  |  |
| 161 | ¡ | P,E,I | 137 |  |  |
| 162 | ¢ | P,E,I | 138 |  |  |
| 163 | £ | P,E,I | 139 |  |  |
| 164 | ¤ | P,E,I | 140 |  |  |
| 165 | ¥ | P,E,I | 141 |  |  |
| 166 | ∣ | P,E,I | 142 |  |  |
| 167 | § | P,E,I | 143 |  |  |
| 168 | ¨ | P,E,I | 144 |  |  |
| 169 | © | P,E,I | 145 |  |  |
| 170 | ª | P,E,I | 146 |  |  |
| 171 | « | P,E,I | 147 |  |  |
| 172 | ¬ | P,E,I | 148 |  |  |
| 173 | — | P,E,I | 149 |  |  |
| 174 | ® | P,E,I | 150 |  |  |
| 175 | – | P,E,I | 151 |  |  |
| 176 | ˙ | P,E,I | 152 |  |  |
| 177 | ± | P,E,I | 153 |  |  |
| 178 | ² | P,E,I | 154 |  |  |
| 179 | ³ | P,E,I | 155 |  |  |
| 180 | ̷ | P,E,I | 156 |  |  |
| 181 | µ | P,E,I | 157 |  |  |
| 182 | ¶ | P,E,I | 158 |  |  |
| 183 | · | P,E,I | 159 |  |  |
| 184 | ¸ | P,E,I | 160 |  |  |
| 185 | ¹ | P,E,I | 161 |  |  |
| 186 | ° | P,E,I | 162 |  |  |
| 187 | » | P,E,I | 163 |  |  |
| 188 | ¼ | P,E,I | 164 |  |  |
| 189 | ½ | P,E,I | 165 |  |  |
| 190 | ¾ | P,E,I | 166 |  |  |
| 191 | ¿ | P,E,I | 167 |  |  |
| 192 | À | A,U,E,I | 65 | 1 | 3 |
| 193 | Á | A,U,E,I | 65 | 1 | 2 |
| 194 | Â | A,U,E,I | 65 | 1 | 4 |
| 195 | Ã | A,U,E,I | 65 | 1 | 6 |
| 196 | Ä | A,U,E,I | 65 | 1 | 5 |
| 197 | Å | A,U,E,I | 65 | 1 | 10 |
| 198 | Æ | A,U,E,I | 65  70 | 1  1 | 1  0 |
| 199 | Ç | A,U,E,I | 67 | 1 | 13 |
| 200 | È | A,U,E,I | 70 | 1 | 3 |
| 201 | É | A,U,E,I | 70 | 1 | 2 |
| 202 | Ê | A,U,E,I | 70 | 1 | 4 |
| 203 | Ë | A,U,E,I | 70 | 1 | 5 |
| 204 | Ì | A,U,E,I | 74 | 1 | 3 |
| 205 | Í | A,U,E,I | 74 | 1 | 2 |
| 206 | Î | A,U,E,I | 74 | 1 | 4 |
| 207 | Ï | A,U,E,I | 74 | 1 | 5 |
| 208 | Đ | A,U,E,I | 69 | 1 | 1 |
| 209 | Ñ | A,U,E,I | 79 | 1 | 6 |
| 210 | Ò | A,U,E,I | 80 | 1 | 3 |
| 211 | Ó | A,U,E,I | 80 | 1 | 2 |
| 212 | Ô | A,U,E,I | 80 | 1 | 4 |
| 213 | Õ | A,U,E,I | 80 | 1 | 6 |
| 214 | Ö | A,U,E,I | 80 | 1 | 5 |
| 215 | × | P,E,I | 168 |  |  |
| 216 | Ø | A,U,E,I | 80 | 1 | 16 |
| 217 | Ù | A,U,E,I | 86 | 1 | 3 |
| 218 | Ú | A,U,E,I | 86 | 1 | 2 |
| 219 | Û | A,U,E,I | 86 | 1 | 4 |
| 220 | Ü | A,U,E,I | 86 | 1 | 5 |
| 221 | Ý | A,U,E,I | 90 | 1 | 2 |
| 222 | Þ | A,U,E,I | 92 | 1 | 1 |
| 223 | ß | A,L,E,I | 84  84 | 0  0 | 1  0 |
| 224 | à | A,L,E,I | 65 | 0 | 3 |
| 225 | á | A,L,E,I | 65 | 0 | 2 |
| 226 | â | A,L,E,I | 65 | 0 | 4 |
| 227 | ã | A,L,E,I | 65 | 0 | 6 |
| 228 | ä | A,L,E,I | 65 | 0 | 5 |
| 229 | å | A,L,E,I | 65 | 0 | 10 |
| 230 | æ | A,L,E,I | 65  70 | 0  0 | 1  0 |
| 231 | ç | A,L,E,I | 67 | 0 | 13 |
| 232 | è | A,L,E,I | 70 | 0 | 3 |
| 233 | é | A,L,E,I | 70 | 0 | 2 |
| 234 | ê | A,L,E,I | 70 | 0 | 4 |
| 235 | ë | A,L,E,I | 70 | 0 | 5 |
| 236 | ì | A,L,E,I | 74 | 0 | 3 |
| 237 | í | A,L,E,I | 74 | 0 | 2 |
| 238 | î | A,L,E,I | 74 | 0 | 4 |
| 239 | ï | A,L,E,I | 74 | 0 | 5 |
| 240 | ð | A,L,E,I | 69 | 0 | 1 |
| 241 | ñ | A,L,E,I | 79 | 0 | 6 |
| 242 | ò | A,L,E,I | 80 | 0 | 3 |
| 243 | ó | A,L,E,I | 80 | 0 | 2 |
| 244 | ô | A,L,E,I | 80 | 0 | 4 |
| 245 | õ | A,L,E,I | 80 | 0 | 6 |
| 246 | ö | A,L,E,I | 80 | 0 | 5 |
| 247 | ÷ | P,E,I | 169 |  |  |
| 248 | ø | A,L,E,I | 80 | 0 | 16 |
| 249 | ù | A,L,E,I | 86 | 0 | 3 |
| 250 | ú | A,L,E,I | 86 | 0 | 2 |
| 251 | û | A,L,E,I | 86 | 0 | 4 |
| 252 | ü | A,L,E,I | 86 | 0 | 5 |
| 253 | ý | A,L,E,I | 90 | 0 | 2 |
| 254 | þ | A,L,E,I | 92 | 0 | 1 |
| 255 | ÿ | A,L,E,I | 90 | 0 | 5 |

Note: unique collation requires that no two rows of this table have identical collation order columns.

**Annex B: Error code translations** (informative)

M1 naked indicator undefined

M2 invalid combination with P fncodatom

M3 $RANDOM seed less than 1

M4 no true condition in $SELECT

M5 lineref less than zero

M6 undefined lvn

M7 undefined gvn

M8 undefined svn

M9 divide by zero

M10 invalid pattern match range

M11 no parameters passed

M12 invalid lineref (negative offset)

M13 invalid lineref (line not found)

M15 undefined index variable

M16 argumented QUIT not allowed

M17 argumented QUIT required

M18 fixed length READ not greater than zero

M19 cannot copy a tree or subtree into itself

M20 method must have formallist

M21 algorithm specification invalid

M22 SET or KILL to ^$GLOBAL when data in global

M23 SET or KILL to ^$JOB for non-existent job number

M24 change to collation algorithm while subscripted local variables defined

M26 non-existent environment

M27 attempt to rollback a transaction that is not restartable

M28 mathematical function, parameter out of range

M29 SET or KILL on ssvn not allowed by implementation

M30 reference to glvn with different collating sequence within a collating algorithm

M31 controlmnemonic used for device without a mnemonicspace selected

M32 controlmnemonic used in user-defined mnemonicspace which as no associated line

M33 SET or KILL to ^$ROUTINE when routine exists

M35 device does not support mnemonicspace

M36 incompatible mnemonicspaces

M37 READ from device identified by the empty string

M38 invalid ssvn subscript

M39 invalid $NAME argument

M40 call-by-reference in JOB actual

M41 invalid LOCK argument within a TRANSACTION

M42 invalid QUIT within a TRANSACTION

M43 invalid range value ($X,$Y)

M44 invalid statement outside of a TRANSACTION

M56 name length limit exceeded

M57 more than one defining occurence of label in routine

M58 too few formal parameters

M75 string length limit exceeded

M90 invalid namevalue

M100 output timeout expired

**Annex C: Metalanguage element dictionary** (Informative)

::= definition

[ ] optional element

| | group of alternate choices

... optional indefinite repetition

actual actual argument

actuallist actual argument list

actualname actual argument name

algoref algorithm reference

alternation alternation

argument argument of a statement

binaryop binary operator

block block

CB close bracket character

charset character set

charsetexpr character set expression

charspec character specification

closeargument CLOSE argument

statement statement

statements statements separated by cs

statementword statement word

comment comment

controlmnemonic control mnemonic

CR carriage return character

cs statement separator

device device

deviceattribute device attribute

devicestatement device statement

devicekeyword device keyword

deviceparam device parameter

deviceparameters device parameters

devicexpr device expression

digit decimal digit

dlabel indirect label (evaluated label)

doargument DO argument

ecode error code

emptystring empty string

entryref entry reference

environment set of distinct names

eoffset error offset

eol end-of-line

eor end-of-routine

exampleargument example argument

examplestatement example statement

exfunc extrinsic function

exp exponent

expr expression

expratom expression atom

expritem expression item

exprtail expression tail

externalroutinename external routine name

externref external reference

extblock external block

extcode external code

extid external identifier

exvar extrinsic variable

fncodatom $FNUMBER code atom

fncode $FNUMBER code

fncodexpr $FNUMBER code expression

fncodp $FNUMBER code P

fncodt $FNUMBER code T

FF form feed character

formallist formal argument list

format I/O format code

forparameter FOR argument

function intrinsic function

functionname function name

glvn global or local variable name

gnamind global name indirection

graphic graphic (character with visible representation)

gvn global variable name

gvnexpr global variable name expression

hangargument HANG argument

ident identification

ifargument IF argument

intexpr expr, value interpreted as an integer

intlit integer literal

jobactuallist JOB actual argument list

jobargument JOB argument

jobenv JOB environment

jobparameters JOB parameters

killargument KILL argument

killarglist KILL argument list

L list (list of)

label label of a line

labelref label reference

leftexpr left expression

leftrestricted left restricted

LF line feed character

libdatatype library data type

library library

libraryelement library element

libraryelementdef library element definition

libraryelementexpr library element expression

libraryexpr library expression

libraryopt library optional flag

libraryparam library element parameter

libraryref library element reference

libraryresult library element result

line line in routine

lineref line reference

lname local name

lnamind local name indirection

lockargument LOCK argument

logicalop logical operator

ls label separator

lvn local variable name

mant mantissa

mergeargument MERGE argument

method method

mnemonicspace mnemonic space

mnemonicspacename mnemonic space name

mnemonicspec mnemonic space specifier

name name

namevalue name value

newargument NEW argument

newsvn NEW svn

noncomma non-comma

nonquote non-quote (any graphic not equal to quote)

nref name reference

numexpr expression, value interpreted numerically

numlit numeric literal

OB open bracket character

openargument OPEN argument

openparameters OPEN parameters

packagename package name

patatom pattern atom

patcode pattern code

patgrp pattern atom group

patnonY pattern non Y

patnonYZ pattern non Y or Z

patnonZ pattern non Z

pattern pattern

place place

postcond post condition

processid process identifier

processparameters process parameters

readargument READ argument

readcount READ count

relation relational operator

repcount repeat count in patatom

restartargument restart argument

rexpratom restricted expression atom

rgvn restricted global variable name

rlvn restricted local variable name

routine routine

routineargument routine argument

routineattribute routine attribute

routinebody routine body

routinehead routine head

routinename routine name

routineref routine reference

routinexpr routine expression

routinekeyword routine keyword

routineparam routine parameter

routineparameters routine parameters

rssvn restricted structured system variable name

setargument SET argument

setdestination SET destination

setev SET error variable

setextract SET $EXTRACT

setleft SET left

setpiece SET $PIECE

setqsub SET $QSUBSCRIPT

SP space character

ssvn structured system variable name

ssvname structured system variable name

ssvnamind structured system variable name indirection

stackcode $STACK code

stackcodexpr $STACK code expression

strconst string constant

strlit string literal

sublit subscript literal

subnonquote subscript non-quote

svn special variable name

system system

systemexpr system expression

textarg $TEXT argument

timeout time-out specification

transparameters transaction parameters

truthop truth operator

tsparam TSTART parameter

tstartargument TSTART argument

tstartkeyword TSTART keyword

tvexpr expr, value interpreted as a truth-value

unaryop unary operator

useargument USE argument

V value (evaluates to)

VB vertical bar character

writeargument WRITE argument

WS white space

xargument EXECUTE argument

**Annex D: Embedded SQL** (Informative)

SQL2 provides a capability for supporting embedded SQL Omega programs. The specification for this is described in ANSI X3.135 (ISO/IEC 9075, 1992) and is partially reproduced here for information purposes. The reader should refer to ANSI X3.135 Section 19 Embedded SQL for the full definition.

**--------------------------------------------------------------**

"**19.1 <embedded SQL host program>**

**...**

**Syntax Rules**

1) An <embedded SQL host program> is a compilation unit that consists of programming language text and SQL text. The programming language text shall conform to the requirements of a specific standard programming language. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s, as defined in this standard.

2) An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> that is contained in an <embedded SQL MUMPS program> shall contain an <SQL prefix> that is "<ampersand>SQL<open paren>". There shall be no <separator> between the <ampersand> and "SQL" nor between "SQL" and the <open paren>.

**...**

3)  **...**

An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> that is contained in an <embedded SQL MUMPS program> shall contain an <SQL terminator> that is a <close paren>.

4) The <token>s comprising an <SQL prefix>, <embedded SQL begin declare>, or <embedded SQL end declare> shall be separated by <space> characters and be specified on one line. Otherwise, the rules for the continuation of lines and tokens from one line to the next and for the placement of host language comments are those of the programming language of the containing <embedded SQL host program>.

**...**

**19.7 <embedded SQL MUMPS program>**

**Function**

Specify an <embedded SQL MUMPS program>

**Format**

<embedded SQL MMUMPS program> ::= !! *See the Syntax Rules.*

<MUMPS host identifier> ::= !! *See the Syntax Rules.*

<MUMPS variable definition> ::=

{ <MUMPS numeric variable> | <MUMPS character variable> } <semicolon>

<MUMPS character variable> ::=

VARCHAR <MUMPS host identifier> <MUMPS length specification>

[ { , <MUMPS host identifier> <MUMPS length specification> }... ]

<MUMPS length specification> ::=

<open paren> <length> <close paren>

<MUMPS numeric variable> ::=

{ INT

| DEC [ { <precisions> [ , <scale> ] ) ]

| REAL }

<MUMPS host identifier> [ { , <MUMPS host identifier> }... ]

**Syntax Rules**

1) An <embedded SQL MUMPS program> is a compilation unit that consists of MUMPS text and SQL text. The MUMPS text shall conform to standard MUMPS. The SQL text shall consist of one or more <embedded SQL statement>s and, optionally, one or more <embedded SQL declare section>s.

2) A <MUMPS host identifier> is any valid MUMPS variable name. A <MUMPS host identifier> shall be contained in an <embedded SQL MUMPS program>.

3) An <embedded SQL statement> may be specified wherever a MUMPS statement may be specified.

4) A <MUMPS variable definition> defines one or more host variables.

5) The <MUMPS character variable> defines a variable-length string. The equivalent SQL data type is VARCHAR whose maximum length is the <length> of the <MUMPS length specification>.

6) INT describes an exact numeric variable. The equivalent SQL data type is INTEGER.

7) DEC describes an exact numeric variable. The <scale> shall not be greater than the <precision>. The equivalent SQL data type is DECIMAL with the same <precision> and <scale>.

8) REAL describes an approximate numeric variable. The equivalent SQL data type is REAL.

9) An <embedded SQL MUMPS program> shall contain either a variable named SQLCODE defined with a datatype of INT or a variable named SQLSTATE defined with a datatype that is VARCHAR with length 5, or both.

**Note:** *SQLSTATE is the preferred status parameter. The SQLCODE status parameter is a deprecated feature that is supported for compatibility with earlier versions of this standard. See Annex D, "Deprecated Features".*

**...**"

**Annex E: Transportability of Omega Software Systems** (informative)

The transfer of routines between machine environments is affected by numerous machine and operating systems factors. A standard transfer format for both routines and data stored within globals cannot at the same time easily cope with the simple and the complex case efficiently, in addition to dealing with the environmental idiosyncrasies. Therefore, the responsibility for the detailed format is left to the transferor.

**1 Routine Transfer Format**

The routine loader routine shall have a form that will load the routines from the transfer medium and will save it in internal format. The save routine creating the transfer medium shall produce the following routine transfer format:

Header-line-1 eol

Header-line-2 eol

routinehead

routine-line eol

.

.

.

eol

routinehead

routine-line eol

.

.

.

eol

[\*\*\*RTN END\*\*\*] eol

In the above structure, routine-line is a string in a format as returned by $TEXT. The two header lines shall be free text and may contain any message the sender wishes to convey to the receiver.

NOTE: Each routine is separated by a blank line (an eol) from the following one. Optionally, either two successive blank lines or the string "\*\*\*RTN END\*\*\*" denotes the end of the file. Eol is defined to be a logical end-of-line record as mutually defined by the sending and receiving environments.

**2 Global Dump Formats**

The global loader shall read and the global dumper shall produce on the transfer medium the following transfer format:

Header-line-1 eol

Header-line-2 eol

Full global reference eol

Data contents eol

.

.

.

Full global reference eol

Data contents eol

eol

[\*\*\*GBL END\*\*\*] eol

Eol is defined to be a logical end-of-line record as mutually defined by the sending and receiving environments.

The full global reference shall conform to a global variable name specification as defined by ANSI/MDC X11.1-1994 section 1, subclause 7.1.2.4. When data contains ASCII control characters, decimal (0-31,127), the user shall be responsible for handling the accurate reconstruction of the data string in the host environment. Subscripts in the full global reference shall not contain the ASCII control characters decimal 0-31 or 127. Optionally, either two successive blank lines or the string "\*\*\*GBL END\*\*\*" denotes the end of the file.

**3 Transfer Media**

If the medium is magnetic tape, it should preferably be 1/2" industry standard 9 track (unlabelled and with the ASCII character set). The eighth bit (i.e., most significant bit) shall be set to zero when transferring 7 bit data. The physical block size shall be preferably 1024 characters but may be any clearly designated integer multiple of this size and preferably use the ANSI X3.27-1987 "D" (unspanned variable length records) format. The reel size should preferably be 7".

Tapes recorded at either 800 bpi, NRZI (ANSI X3.22-1990) or 1600 or 6250 bpi phase-encoded (ANSI X3.39-1992) are recommended for current systems.

**Annex F: X3.64 Controlmnemonics** (informative)

Control- Control-

mnemonic Control Function mnemonic Control Function

APC Application Program Statement

CBT Cursor Backward Tabulation

CCH Cancel Character

CHA Cursor Horizontal Absolute

CHT Cursor Horizontal Tabulation

CNL Cursor Next Line

CPL Cursor Preceding Line

CPR Cursor Position Report

CTC Cursor Tabulation Control

CUB Cursor Backward

CUD Cursor Down

CUF Cursor Forward

CUP Cursor Position

CUU Cursor Up

CVT Cursor Vertical Tabulation

DA Device Attributes

DAQ Define Area Qualification

DCH Delete Character

DCS Device Control String

DL Delete Line

DMI Disable Manual Input

DSR Device Status Report

EA Erase in Area

ECH Erase Character

ED Erase in Display

EF Erase in Field

EL Erase in Line

EMI Enable Manual Input

EPA End of Protected Area

ESA End of Selected Area

FNT Font Selection

GSM Graphic Size Modification

GSS Graphic Size Selection

HPA Horizontal Position Absolute

HPR Horizontal Position Relative

HTJ Horizontal Tab with Justify

HTS Horizontal Tabulation Set

HVP Horizontal and Vertical Position

ICH Insert Character

IL Insert Line

IND Index

INT Interrupt

JFY Justify

MC Media Copy

MW Message Waiting

NEL Next Line

NP Next Page

OSC Operating System Statement

PLD Partial Line Down

PLU Partial Line Up

PM Privacy Message

PP Preceding Page

PU1 Private Use One

PU2 Private Use Two

QUAD QUAD

REP Repeat

RI Reverse Index

RIS Reset to Initial State

RM Reset Mode

SEM Select Editing Extent Mode

SGR Select Graphic Rendition

SL Scroll Left

SM Set Mode

SPA Start of Protected Area

SPI Spacing Increment

SR Scroll Right

SS2 Single Shift Two

SS3 Single Shift Three

SSA Start of Selected Area

ST String Terminator

STS Set Transmit State

SU Scroll Up

TBC Tabulation Clear

TSS Thin Space Specification

VPA Vertical Position Absolute

VPR Vertical Position Relative

VTS Vertical Tabulation Set

**Annex G: charset JIS90** (informative)

(This is a partial English reproduction of the JIS90 charset. The reader should refer to JIS X0201-1990 and JIS X0208-1990 for the full definition.)

**1. charset JIS90**

The charset JIS90 is defined using the JIS X0201-1990 8-bit Code and the JIS X0208-1990 2-Byte ode for Information Interchange.

**2. JIS X0201-1990**

In JIS X0201-1990, the values of decimal and character are the same as those from ASCII (X3.4-1990) in the range between decimal 0-127, except decimal 92 which represents "¥" (yen) instead of "\" and Decimal 126 which represents "\_" (overline) instead of "~" (tilde).

The patcodes defined in charset M as A, C, E, L, N, P, and U apply in the same way in the range of decimal 0-127.

In the decimal range between 161 and 223, the values represent 8-bit katakana characters.

**3. JIS X0208-1990**

In JIS X0208-1990, the relation of decimal and character is obtained as following. Let C1 and C2 be the decimal values of the 1st byte and the 2nd byte code for character, then the range of decimal code for both C1 and C2 is [33,127] and the decimal value of the character is C1\*256+C2. Let *n* be a decimal and if there is no character assigned for *n* in JIS X0208, then $C(*n*) is a space as exemplified by $C(8481).

**4. Pattern Codes**

Patcodes E and (ka, $C(182)) apply for the characters in the decimal range 161-223. Patcodes E and (zen, $C(16692)) apply for the characters in the decimal range 8481-32382.

**5. Characters used in names**

Characters in the charset JIS90 except $C(8481) may be defined as ident.

**6. Collation**

The collation scheme of charset JIS90 is ordered by the $A value of the character, whitin each of JIS X0201-1990 and X0208-1990.

**Annex H: Example Code for Library Functions** (informative)

The code in this annex is an example of a possible implementation of these library functions. Implementors are encouraged to provide implementations that offer a better efficiency as well as greater accuracy.

**1 CHARACTER Library**

**1.1 COLLATE**

**1.2 COMPARE**

**2 MATH Library**

**2.1 ABS**

ABS(X) QUIT $TRANSLATE(+X,"-")

**2.2 ARCCOS**

ARCCOS(X,PREC) ;

NEW L,LIM,K,SIG,SIGS

IF X<-1 SET $ECODE=",M28,"

IF X>1 SET $ECODE=",M28,"

SET PREC=$GET(PREC,11)

IF $TRANSLATE(X,"-")=1 SET VALUE=0 QUIT

SET SIG=$SELECT(X<0:-1,1:1),VALUE=1-(X\*X)

SET X=$%SQRT^MATH(VALUE)

IF $TRANSLATE(X,"-")=1 DO QUIT

. SET VALUE=$%PI^MATH()/2\*X

IF X>.9 DO QUIT

. SET SIGS=$SELECT(X<0:-1,1:1)

. SET VALUE=1/(1/X/X-1)

. SET X=$%SQRT^MATH(VALUE)

. SET VALUE=$%ARCTAN^MATH(X,10)\*SIGS

SET (VALUE,L)=X

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 DO QUIT:($TRANSLATE(L,"-")<LIM)

. SET L=L\*X\*X\*(K-2)/(K-1)/K,VALUE=VALUE+L

QUIT $SELECT(SIG<0:$%PI^MATH()-VALUE,1:VALUE)

**2.3 ARCCOSH**

ARCCOSH(X,PREC) ;

IF X<1 SET $ECODE=",M28,"

SET PREC=$GET(PREC,11)

NEW SQ

SET SQ=$%SQRT^MATH(X\*X-1,PREC)

QUIT $%LOG^MATH(X+SQ,PREC)

**2.4 ARCCOT**

ARCCOT(X,PREC) ;

SET PREC=$GET(PREC,11)

SET X=1/X

QUIT $%ARCTAN^MATH(X,PREC)

**2.5 ARCCOTH**

ARCCOTH(X,PREC) ;

NEW L1,L2

SET PREC=$GET(PREC,11)

SET L1=$%LOG^MATH(X+1,PREC)

SET L2=$%LOG^MATH(X-1,PREC)

QUIT L1-L2/2

**2.6 ARCCSC**

ARCCSC(X,PREC) ;

SET PREC=$GET(PREC,11)

SET X=1/X

QUIT $%ARCSIN^MATH(X,PREC)

**2.7 ARCSEC**

ARCSEC(X,PREC) ;

SET PREC=$GET(PREC,11)

SET X=1/X

QUIT $%ARCCOS^MATH(X,PREC)

**2.8 ARCSIN**

ARCSIN(X,PREC) ;

NEW L,LIM,X,SIGS,VALUE

SET PREC=$GET(PREC,11)

IF $TRANSLATE(X,"-")=1 DO QUIT

. SET VALUE=$%PI^MATH()/2\*X

IF X>.99999 DO QUIT

. SET SIGS=$SELECT(X<0:-1,1:1)

. SET VALUE=1/(1/X/X-1)

. SET X=$%SQRT^MATH(VALUE)

. SET VALUE=$%ARCTAN^MATH(X,10)\*SIGS

SET (VALUE,L)=X

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 DO QUIT:($TRANSLATE(L,"-")<LIM)

. SET L=L\*X\*X\*(K-2)/(K-1)\*(K-2)/K,VALUE=VALUE+L

QUIT VALUE

**2.9 ARCSINH**

ARCSINH(X,PREC) ;

IF X<1 SET $ECODE=",M28,"

SET PREC=$GET(PREC,11)

NEW SQ

SET SQ=$%SQRT^MATH(X\*X+1,PREC)

QUIT $%LOG^MATH(X+SQ,PREC)

**2.10 ARCTAN**

ARCTAN(X,PREC) ;

NEW FOLD,HI,L,LIM,LO,K,SIGN,SIGS,SIGT,VALUE

SET PREC=$GET(PREC,11)

SET LO=.0000000001,HI=9999999999

SET SIGT=$SELECT(X<0:-1,1:1)

SET X=$SELECT(X<LO:LO,X>HI:HI,1:X)

SET FOLD=$SELECT(X'<1:0,1:1)

SET L=X,VALUE=$%PI^MATH()/2-(1/X),SIGN=1

IF X<1.3 DO QUIT

. SET X=$SELECT(FOLD:1/X,1:X),VALUE=1/((1/X/X)+1)

. SET $%SQRT^MATH(VALUE)

. IF $TRANSLATE(X,"-")=1 DO QUIT

. . SET VALUE=$%PI^MATH()/2\*X

. IF X>.9 DO QUIT

. . SET SIGS=$SELECT(X<0:-1,1:1)

. . SET VALUE=1/(1/X/X-1)

. . SET X=$%SQRT^MATH(VALUE)

. . SET VALUE=$$ARCTAN(X,10)

. . SET VALUE=VALUE\*SIGS

. SET (VALUE,L)=X

. SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

. FOR K=3:2 DO QUIT:($TRANSLATE(L,"-")<LIM)

. . SET L=L\*X\*X\*(K-2)/(K-1)\*(K-2)/K, VALUE=VALUE+L

. SET VALUE=$SELECT(SIGT<1:-VALUE,1:VALUE)

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 DO QUIT:($TRANSLATE(1/L,"-")<LIM)

. SET L=L\*X\*X,VALUE=VALUE+(1/(K\*L)\*SIGN),SIGN=SIGN\*-1

SET VALUE=$SELECT(FOLD:$%PI^MATH()/2-VALUE,1:VALUE)

SET VALUE=$SELECT(SIGT<1:-VALUE,1:VALUE)

QUIT VALUE

**2.11 ARCTANH**

ARCTANH(X,PREC) ;

IF X<-1 SET $ECODE=",M28,"

IF X>1 SET $ECODE=",M28,"

QUIT $%LOG^MATH(1+X/(1-X),$GET(PREC,11))/2

**2.12 CABS**

CABS(Z) ;

NEW ZRE,ZIM

SET ZRE=+Z,ZIM=+$P(Z,"%",2)

QUIT $%SQRT^MATH(ZRE\*ZRE+(ZIM\*ZIM))

**2.13 CADD**

CADD(X,Y) ;

NEW XRE,XIM,YRE,YIM

SET XRE=+X,XIM=+$P(X,"%",2)

ZET YRE=+Y,YIM=+$P(Y,"%",2)

QUIT XRE+YRE\_"%"\_(XIM+YIM)

**2.14 CCOS**

CCOS(Z,PREC) ;

NEW E1,E2,IA

SET PREC=$GET(PREC,11)

SET IA=$%CMUL^MATH(Z,"0%1")

SET E1=$%CEXP^MATH(IA,PREC)

SET IA=-IA\_"%"\_(-$PIECE(IA,"%",2))

SET E2=$%CEXP^MATH(IA,PREC)

SET IA=$%CADD^MATH(E1,E2)

QUIT $%CMUL^MATH(IA,"0.5%0")

**2.15 CDIV**

CDIV(X,Y) ;

NEW D,IM,RE,XIM,XRE,YIM,YRE

SET XRE=+X,XIM=+$P(X,"%",2)

SET YRE=+Y,YIM=+$P(Y,"%",2)

SET D=YRE\*YRE+(YIM\*YIM)

SET RE=XRE\*YRE+(XIM\*YIM)/D

SET IM=XIM\*YRE-(XRE\*YIM)/D

QUIT RE\_"%"\_IM

**2.16 CEXP**

CEXP(Z,PREC) ;

NEW R,ZIM,ZRE

SET PREC=$GET(PREC,11)

SET ZRE=+Z,ZIM=+$P(Z,"%",2)

SET R=$%EXP^MATH(ZRE,PREC)

QUIT R\*$%COS^MATH(ZIM,PREC)\_"%"\_(R\*$%SIN^MATH(ZIM,PREC))

**2.17 CLOG**

CLOG(Z,PREC) ;

NEW ABS,ARG,ZIM,ZRE

SET PREC=$GET(PREC,11)

SET ABS=$%CABS^MATH(Z)

SET ZRE=+Z,ZIM=+$P(Z,"%",2)

SET ARG=$%ARCTAN^MATH(ZIM,ZRE,PREC)

QUIT $%LOG^MATH(ABS,PREC)\_"%"\_ARG

**2.18 CMUL**

CMUL(X,Y) ;

NEW XIM,XRE,YIM,YRE

SET XRE=+X,XIM=+$P(X,"%",2)

SET YRE=+Y,YIM=+$P(Y,"%",2)

QUIT XRE\*YRE-(XIM\*YIM)\_"%"\_(XRE\*YIM+(XIM\*YRE))

**2.19 COMPLEX**

COMPLEX(X) QUIT +X\_"%0"

**2.20 CONJUG**

CONJUG(Z) ;

NEW ZIM,ZRE

SET ZRE=+Z,ZIM=+$P(Z,"%",2)

QUIT ZRE\_"%"\_(-ZIM)

**2.21 COS**

COS(X,PREC) ;

NEW L,LIM,K,SIGN,VALUE

SET:X[":" X=$%DMSDEC^MATH(X,12)

SET PREC=$GET(PREC,11)

SET X=X#(2\*$%PI^MATH())

SET (VALUE,L)=1,SIGN=-1

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=2:2 DO QUIT:($TRANSLATE(L,"-")<LIM) SET SIGN=SIGN\*-1

. SET L=L\*X\*X/(K-1\*K),VALUE=VALUE+(SIGN\*L)

QUIT VALUE

**2.22 COSH**

COSH(X,PREC) ;

NEW E,F,I,P,R,T,XX

SET PREC=$GET(PREC,11)+1

SET @("E=1E-"\_PREC)

SET XX=X\*X,F=1,(P,R,T)=1,I=1

FOR SET T=T\*XX,F=I+1\*I\*F,R=T/F+R,P=P-R/R,I=I+2 IF -E<P,P<E QUIT

QUIT R

**2.23 COT**

COT(X,PREC) ;

NEW C,L,LIM,K,SIGN,VALUE

SET:X[":" X=$%DMSDEC^MATH(X,12)

SET PREC=$GET(PREC,11)

SET (VALUE,L)=1,SIGN=-1

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=2:2 DO QUIT:($TRANSLATE(L,"-")<LIM) SET SIGN=SIGN\*-1

. SET L=L\*X\*X/(K-1\*K),VALUE=VALUE+(SIGN\*L)

SET C=VALUE

SET X=X#(2\*$%PI^MATH())

SET (VALUE,L)=1,SIGN=-1

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 DO QUIT:($TRANSLATE(L,"-")<LIM) SET SIGN=SIGN\*-1

. SET L=L/(K-1)\*X/K\*X,VALUE=VALUE+(SIGN\*L)

IF 'VALUE QUIT "INFINITE"

QUIT VALUE=C/VALUE

**2.24 COTH**

COTH(X,PREC) ;

NEW SINH

IF 'X QUIT "INFINITE"

SET PREC=$GET(PREC,11)

SET SINH=$%SINH^MATH(X,PREC)

IF 'SINH QUIT "INFINITE"

QUIT $%COSH^MATH(X,PREC)/SINH

**2.25 CPOWER**

CPOWER(Z,N,PREC) ;

NEW AR,NIM,NRE,PHI,PI,R,RHO,TH,ZIM,ZRE

SET ZRE=+Z,ZIM=+$P(Z,"%",2)

SET NRE=+N,NIM=+$P(N,"%",2)

SET PREC=$GET(PREC,11)

IF 'ZRE,'ZIM,'NRE,'NIM SET $ECODE=",M28"

IF 'ZRE,'ZIM QUIT "0%0"

SET PI=$%PI^MATH()

SET R=$%SQRT^MATH(ZRE\*ZRE+(ZIM\*ZIM),PREC)

IF ZRE SET TH=$%ARCTAN^MATH(ZIM,ZRE,PREC) ?????

ELSE SET TH=$SELECT(ZRE>0:PI/2,1:-PI/2)

SET RHO=$%LOG^MATH(R,PREC)

SET AR=$%EXP^MATH(RHO\*NRE-(TH\*NIM),PREC)

SET PHI=RHO\*NIM+(NRE\*TH)

QUIT AR\*$%COS^MATH(PHI,PREC)\_"%"\_(AR\*$%SIN^MATH(PHI,PREC))

**2.26 CSC**

CSC(X,PREC) ;

NEW L,LIM,K,SIGN,VALUE

SET:X[":" X=$%DMSDEC^MATH(X,12)

SET PREC=$GET(PREC,11)

SET X=X#(2\*$%PI^MATH())

SET (VALUE,L)=1,SIGN=-1

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 DO QUIT:($TRANSLATE(L,"-")<LIM) SET SIGN=SIGN\*-1

. SET L=L/(K-1)\*X/K\*X,VALUE=VALUE+(SIGN\*L)

IF 'VALUE QUIT "INFINITE"

QUIT 1/VALUE

**2.27 CSCH**

CSCH(X,PREC) QUIT 1/$%SINH^MATH(X,$GET(PREC,11))

**2.28 CSIN**

CSIN(Z,PREC) ;

NEW IA,E1,E2

SET PREC=$GET(PREC,11)

SET IA=$%CMUL^MATH(Z,"0%1")

SET E1=$%CEXP^MATH(IA,PREC)

SET IA=-IA\_"%"\_(-$PIECE(IA,"%",2))

SET E2=$%CEXP^MATH(IA,PREC)

SET IA=$%CSUB^MATH(E1,E2)

SET IA=$%CMUL^MATH(IA,".5%0")

QUIT $%CMUL^MATH("0%-1",IA)

**2.29 CSUB**

CSUB(X,Y) ;

NEW XIM,XRE,YIM,YRE

SET XRE=+X,XIM=+$P(X,"%",2)

SET YRE=+Y,YIM=+$P(Y,"%",2)

QUIT XRE-YRE\_"%"\_(XIM-YIM)

**2.30 DECDMS**

DECDMS(X,PREC) ;

SET PREC=$GET(PREC,5)

SET X=X#360\*3600

SET X=+$JUSTIFY(X,0,$SELECT((PREC-$LENGTH(X\1))'<0:PREC-$LENGTH(X\1),1:0))

QUIT X\3600\_":"\_(X\60#60)\_":"\_(X#60)

**2.31 DEGRAD**

DEGRAD(X) QUIT X\*3.14159265358979/180

**2.32 DMSDEC**

DMSDEC(X) QUIT $PIECE(X,":")+($PIECE(X,":",2)/60)+($PIECE(X,":",3)/3600)

**2.33 E**

E() QUIT 2.71828182845905

**2.34 EXP**

EXP(X,PREC) ;

NEW L,LIM,K,VALUE

SET PREC=$GET(PREC,11)

SET L=X,VALUE=X+1

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=2:1 SET L=L\*X/K,VALUE=VALUE+L QUIT:($TRANSLATE(L,"-")<LIM)

QUIT VALUE

**2.35 LOG**

LOG(X,PREC) ;

NEW L,LIM,M,N,K,VALUE

IF X'>0 SET $ECODE=",M28,"

SET PREC=$GET(PREC,11)

SET M=1

IF X>0 FOR N=0:1 QUIT:(X/M)<10 SET M=M\*10

IF X<1 FOR N=0:-1 QUIT:(X/M)>.1 SET M=M\*.1

SET X=X/M

SET X=(X-1)/(X+1),(VALUE,L)=X

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 SET L=L\*X\*X,M=L/K,VALUE=M+VALUE SET:M<0 M=-M QUIT:M<LIM

SET VALUE=VALUE\*2+(N\*2.30258509298749)

QUIT VALUE

**2.36 LOG10**

LOG10(X,PREC) ;

NEW L,LIM,M,N,K,VALUE

IF X'>0 SET $ECODE=",M28,"

SET PREC=$GET(PREC,11)

SET M=1

IF X>0 FOR N=0:1 QUIT:(X/M)<10 SET M=M\*10

IF X<1 FOR N=0:-1 QUIT:(X/M)>.1 SET M=M\*.1

SET X=X/M

SET X=(X-1)/(X+1),(VALUE,L)=X

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 SET L=L\*X\*X,M=L/K,VALUE=M+VALUE SET:M<0 M=-M QUIT:M<LIM

SET VALUE=VALUE\*2+(N\*2.30258509298749)

QUIT VALUE/2.30258509298749

**2.37 MTXADD**

MTXADD(A,B,R,ROWS,COLS) ; Add A[ROWS,COLS] to B[ROWS,COLS],

; result goes to R[ROWS,COLS]

IF $DATA(A)<10 QUIT 0

IF $DATA(B)<10 QUIT 0

IF $GET(ROWS)<1 QUIT 0

IF $GET(COLS)<1 QUIT 0

;

NEW ROW,COL,ANY

FOR ROW=1:1:ROWS FOR COL=1:1:COLS DO

. KVALUE R(ROW,COL) SET ANY=0

. SET:$DATA(A(ROW,COL))#2 ANY=1

. SET:$DATA(B(ROW,COL))#2 ANY=1

. SET:ANY R(ROW,COL)=$GET(A(ROW,COL))+$GET(B(ROW,COL))

. QUIT

QUIT 1

**2.38 MTXCOF**

MTXCOF(A,I,K,N) ; Compute cofactor for element [I,K] in matrix A[N,N]

NEW T,R,C,RR,CC

SET CC=0 FOR CC=1:1:N DO:C'=K

. SET CC=CC+1,RR=0

. FOR R=1:1:N SET:R'=I RR=RR+1,T(RR,CC)=$GET(A(R,C))

. QUIT

QUIT $%MTXDET^MATH(.T,N-1)

**2.39 MTXCOPY**

MTXCOPY(A,R,ROWS,COLS) ; Copy A[ROWS,COLS] to R[ROWS,COLS]

IF $DATA(A)<10 QUIT 0

IF $GET(ROWS)<1 QUIT 0

IF $GET(COLS)<1 QUIT 0

;

NEW ROW,COL

FOR ROW=1:1:ROWS FOR COL=1:1:COLS DO

. KVALUE R(ROW,COL)

. SET:$DATA(A(ROW,COL))#2 R(ROW,COL)=A(ROW,COL)

. QUIT

QUIT 1

**2.40 MTXDET**

MTXDET(A,N) ; Compute determinant of matrix A[N,N]

IF $DATA(A)<10 QUIT ""

IF $GET(N)<1 QUIT ""

;

; First the simple cases

;

IF N=1 QUIT $GET(A(1,1))

IF N=2 QUIT $GET(A(1,1))\*$GET(A(2,2))-($GET(A(1,2))\*$GET(A(2,1)))

;

NEW DET,I,SIGN

;

; Det A = sum (K=1:N) element (I,K) \* cofactor [I,K]

;

SET DET=0,SIGN=1

FOR I=1:1:N DO

. SET DET=$GET(A(1,I))\*$%MTXCOF^MATH(.A,1,I,N)\*SIGN+DET

. SET SIGN=-SIGN

. QUIT

QUIT DET

**2.41 MTXEQU**

MTXEQU(A,B,R,N,M) ; Solve matrix equation A [M,M] \* R [M,N] = B [M,N]

IF $GET(M)<1 QUIT ""

IF $GET(N)<1 QUIT ""

IF '$%MTXDET^MATH(.A,M) QUIT 0

;

NEW I,I1,J,J1,J2,K,L,T,T1,T2,TEMP,X

;

SET X=$%MTXCOPY^MATH(.A,.T,N,N)

SET X=$%MTXCOPY^MATH(.B,.R,N,M)

;

; Reduction of matrix A

; Steps of reduction are counted by index K

;

FOR K=1:1:N-1 DO

. ;

. ; Search for largest coefficient of T (denoted by TEMP)

. ; in first column of reduced system

. ;

. SET TEMP=0,J2=K

. FOR J1=K:1:N DO

. . QUIT:$TRANSLATE($GET(T(J1,K)),"-")'>$TRANSLATE(TEMP,"-")

. . SET TEMP=T(J1,K),J2=J1

. . QUIT

. ;

. ; Exchange row number K with row number J2, if necessary

. ;

. DO:J2'=K

. . ;

. . FOR J=K:1:N DO

. . . SET T1=$GET(T(K,J)),T2=$GET(T(J2,J)) KILL T(K,J),T(J2,J)

. . . IF T1'="" SET T(J2,J)=T1

. . . IF T2'="" SET T(K,J)=T2

. . . QUIT

. . FOR J=1:1:M DO

. . . SET T1=$GET(R(K,J)),T2=$GET(R(J2,J)) KILL R(K,J),R(J2,J)

. . . IF T1'="" SET R(J2,J)=T1

. . . IF T2'="" SET R(K,J)=T2

. . . QUIT

. . QUIT

. ;

. ; Actual translation

. ;

. FOR I=K+1:1:N DO

. . J=K+1:1:N DO

. . . QUIT:'$GET(T(K,K))

. . . SET T(I,J)=-$GET(T(K,J))\*$GET(T(I,K))/T(K,K)+$GET(T(I,J))

. . . QUIT

. . FOR J=1:1:M DO

. . . QUIT:'$GET(T(K,K))

. . . SET R(I,J)=-$GET(R(K,J))\*$GET(T(I,K))/T(K,K)+$GET(R(I,J))

. . . QUIT

. . QUIT

. QUIT

;

; Backsubstitution

;

FOR J=1:1:M DO

. IF $GET(T(N,N)) SET R(N,J)=$GET(R(N,J))/T(N,N)

. IF N-1>0 FOR I1=1:1:N-1 DO

. . SET I=N-I1

. . FOR L=I+1:1:N DO

. . . SET R(I,J)=-$GET(T(I,L))\*$GET(R(L,J))+$GET(R(I,J))

. . . QUIT

. . IF $GET(T(I,I)) SET R(I,J)=$GET(R(I,J))/$GET(T(I,I))

. . QUIT

. QUIT

QUIT $%MTXDET^MATH(.R,?\*\*\*\*\*?)

**2.42 MTXINV**

MTXINV(A,R,N) ; Invert A[N,N], result goes to R[N,N]

IF $DATA(A)<10 QUIT 0

IF $GET(N)<1 QUIT 0

;

NEW T,X

SET X=$%MTXUNIT^MATH(.T,N)

QUIT $%MTXEQU^MATH(.A,.T,.R,N,N)

**2.43 MTXMUL**

MTXMUL(A,B,R,M,L,N) ; Multiply A[M,L] by B[M,L], result goes to R[M,N]

IF $DATA(A)<10 QUIT 0

IF $DATA(B)<10 QUIT 0

IF $GET(L)<1 QUIT 0

IF $GET(M)<1 QUIT 0

IF $GET(N)<1 QUIT 0

;

NEW I,J,K,SUM,ANY

FOR I=1:1:M FOR J=1:1:N DO

. SET (SUM,ANY)=0

. KVALUE R(I,J)

. FOR K=1:1:L DO

. . SET:$DATA(A(I,K))#2 ANY=1

. . SET:$DATA(B(K,J))#2 ANY=1

. . SET SUM=$GET(A(I,K))\*$GET(B(K,J))+SUM

. . QUIT

. SET:ANY R(I,J)=SUM

. QUIT

QUIT 1

**2.44 MTXSCA**

MTXSCA(A,R,ROWS,COLS,S) ; Multiply A[ROWS,COLS] with the scalar S,

; result goes to R[ROWS,COLS]

IF $DATA(A)<10 QUIT 0

IF $GET(ROWS)<1 QUIT 0

IF $GET(COLS)<1 QUIT 0

IF '($DATA(S)#2) QUIT 0

;

NEW ROW,COL

FOR ROW=1:1:ROWS FOR COL=1:1:COLS DO

. KVALUE R(ROW,COL)

. SET:$DATA(A(ROW,COL))#2 R(ROW,COL)=A(ROW,COL)\*S

. QUIT

QUIT 1

**2.45 MTXSUB**

MTXSUB(A,B,R,ROWS,COLS) ; Subtract B[ROWS,COLS] from A[ROWS,COLS],

; result goes to R[ROWS,COLS]

IF $DATA(A)<10 QUIT 0

IF $DATA(B)<10 QUIT 0

IF $GET(ROWS)<1 QUIT 0

IF $GET(COLS)<1 QUIT 0

;

NEW ROW,COL,ANY

FOR ROW=1:1:ROWS FOR COL=1:1:COLS DO

. KVALUE R(ROW,COL) SET ANY=0

. SET:$DATA(A(ROW,COL))#2 ANY=1

. SET:$DATA(B(ROW,COL))#2 ANY=1

. SET:ANY R(ROW,COL)=$GET(A(ROW,COL))-$GET(B(ROW,COL))

. QUIT

QUIT 1

**2.46 MTXTRP**

MTXTRP(A,R,M,N) ; Transpose A[M,N], result goes to R[M,N]

IF $DATA(A)<10 QUIT 0

IF $GET(M)<1 QUIT 0

IF $GET(N)<1 QUIT 0

;

NEW I,J,K,D1,V1,D2,V2

FOR I=1:1:M+N-1 FOR J=1:1:I+1\2 DO

. SET K=I-J+1

. IF K=J DO QUIT

. . SET V1=$GET(A(J,J)),D1=$DATA(A(J,J))#2

. . IF J'>N,J'>M KVALUE R(J,J) SET:D1 R(J,J)=V1

. . QUIT

. ;

. SET V1=$GET(A(K,J)),D1=$DATA(A(K,J))#2

. SET V2=$GET(A(J,K)),D2=$DATA(A(J,K))#2

. IF K'>M,J'>N KVALUE R(K,J) SET:D2 R(K,J)=V2

. IF J'>M,K'>N KVALUE R(J,K) SET:D1 R(J,K)=V1

. QUIT

QUIT 1

**2.47 MTXUNIT**

MTXUNIT(R,N,SPARSE) ; Create a unit matrix R[N,N]

IF $GET(N)<1 QUIT 0

;

NEW ROW,COL

FOR ROW=1:1:N FOR COL=1:1:N DO

. KVALUE R(ROW,COL)

. IF $GET(SPARSE) QUIT:ROW'=COL

. SET R(ROW,COL)=$SELECT(ROW=COL:1,1:0)

. QUIT

QUIT 1

**2.37 PI**

PI() QUIT 3.14159265358979

**2.38 RADDEG**

RADDEG(X) QUIT X\*180/3.14159265358979

**2.39 SEC**

SEC(X,PREC) ;

NEW L,LIM,K,SIGN,VALUE

SET:X[":" X=$%DMSDEC^MATH(X,12)

SET PREC=$GET(PREC,11)

SET X=X#(2\*$%PI^MATH())

SET (VALUE,L)=1,SIGN=-1

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 DO QUIT:($TRANSLATE(L,"-")<LIM) SET SIGN=SIGN\*-1

. SET L=L\*X\*X/(K-1\*K),VALUE=VALUE+(SIGN\*L)

IF 'VALUE QUIT "INFINITE"

QUIT 1/VALUE

**2.40 SECH**

SECH(X,PREC) QUIT 1/$%COSH^MATH(X,$GET(PREC,11))

**2.41 SIGN**

SIGN(X) QUIT $SELECT(X<0:-1,X>0:1,1:0)

**2.42 SIN**

SIN(X,PREC) ;

NEW L,LIM,K,SIGN,VALUE

SET:X[":" X=$%DMSDEC^MATH(X,12)

SET PREC=$GET(PREC,11)

SET X=X#(2\*$%PI^MATH())

SET (VALUE,L)=1,SIGN=-1

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 DO QUIT:($TRANSLATE(L,"-")<LIM) SET SIGN=SIGN\*-1

. SET L=L/(K-1)\*X/K\*X,VALUE=VALUE+(SIGN\*L)

QUIT VALUE

**2.43 SINH**

SINH(X,PREC) ;

NEW E,F,I,P,R,T,XX

SET PREC=$GET(PREC,11)+1

SET @("E=1E-"\_PREC)

SET XX=X\*X,F=1,(P,R,T)=X,I=2

FOR SET T=T\*XX,F=I+1\*I\*F,R=T/F+R,P=P-R/R,I=I+2 IF -E<P,P<E QUIT

QUIT R

**2.44 SQRT**

SQRT(X,PREC) ;

IF X<0 SET $ECODE=",M28,"

IF X=0 QUIT 0

IF X<1 QUIT 1/$%SQRT^MATH(1/X)

NEW P,R,E

SET PREC=$GET(PREC,11)+1

SET @(E="1E-"\_PREC)

SET R=X

FOR SET P=R,R=X/R+R/2,P=P-R/R IF -E<P,P<E QUIT

QUIT R

**2.45 TAN**

TAN(X,PREC) ;

NEW L,LIM,K,S,SIGN,VALUE

SET:X[":" X=$%DMSDEC^MATH(X,12)

SET PREC=$GET(PREC,11)

SET X=X#(2\*$%PI^MATH())

SET (VALUE,L)=1,SIGN=-1

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=3:2 DO QUIT:($TRANSLATE(L,"-")<LIM) SET SIGN=SIGN\*-1

. SET L=L/(K-1)\*X/K\*X,VALUE=VALUE+(SIGN\*L)

SET S=VALUE

SET X=X#(2\*$%PI^MATH())

SET (VALUE,L)=1,SIGN=-1

SET LIM=$SELECT((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"\_LIM)

FOR K=2:2 DO QUIT:($TRANSLATE(L,"-")<LIM) SET SIGN=SIGN\*-1

. SET L=L\*X\*X/(K-1\*K),VALUE=VALUE+(SIGN\*L)

IF 'VALUE QUIT "INFINITE"

QUIT S/VALUE

**2.46 TANH**

TANH(X,PREC) QUIT $%SINH^MATH(X,$GET(PREC,11))/$%COSH^MATH(X,$GET(PREC,11))

**3 STRING Library**

**3.1 PRODUCE**

PRODUCE(IN,SPEC,MAX) ;

NEW VALUE,AGAIN,P1,P2,I,COUNT

SET VALUE=IN,COUNT=0

FOR DO QUIT:'AGAIN

. SET AGAIN=0

. SET I=""

. FOR SET I=$ORDER(SPEC(I)) QUIT:I="" DO QUIT:COUNT<0

. . QUIT:$GET(SPEC(I,1))=""

. . QUIT:'($DATA(SPEC(I,2))#2)

. . FOR QUIT:VALUE'[SPEC(I,1) DO QUIT:COUNT<0

. . . SET P1=$PIECE(VALUE,SPEC(I,1),1)

. . . SET P2=$PIECE(VALUE,SPEC(I,2),2,$LENGTH(VALUE))

. . . SET VALUE=P1\_SPEC(I,2)\_P2,AGAIN=1

. . . SET COUNT=COUNT+1

. . . IF $DATA(MAX),COUNT>MAX SET COUNT=-1,AGAIN=0

. . . QUIT

. . QUIT

. QUIT

QUIT VALUE

**3.2 REPLACE**

REPLACE(IN,SPEC) ;

NEW L,MASK,K,I,LT,F,VALUE

SET L=$LENGTH(IN),MASK=$JUSTIFY("",L)

SET I="" FOR SET I=$ORDER(SPEC(I)) QUIT:I="" DO

. QUIT:'($DATA(SPEC(I,1))#2)

. QUIT:SPEC(I,1)=""

. QUIT:'($DATA(SPEC(I,2))#2)

. SET LT=$LENGTH(SPEC(I,1))

. SET F=0 FOR SET F=$FIND(IN,SPEC(I,1),F) QUIT:F<1 DO

. . QUIT:$EXTRACT(MASK,F-LT,F-1)["X"

. . SET VALUE(F-LT)=SPEC(I,2)

. . SET $EXTRACT(MASK,F-LT,F-1)=$TRANSLATE($JUSTIFY("",LT)," ","X")

. . QUIT

. QUIT

SET VALUE="" FOR K=1:1:L DO

. IF $EXTRACT(MASK,K)=" " SET VALUE=VALUE\_$EXTRACT(IN,K) QUIT

. SET:$DATA(VALUE(K)) VALUE=VALUE\_VALUE(K)

. QUIT

QUIT VALUE

Index

$%ABS^MATH 47

$%ARCCOSH^MATH 47

$%ARCCOT^MATH 47

$%ARCCOTH^MATH 47

$%ARCCSC^MATH 47, 48

$%ARCOS^MATH 47

$%ARCSEC^MATH 48

$%ARCSIN^MATH 48, 127

$%ARCSINH^MATH 48

$%ARCTAN^MATH 48, 126, 127, 129, 130

$%ARCTANH^MATH 48

$%CABS^MATH 48, 129

$%CADD^MATH 48, 49, 128

$%CCOS^MATH 49

$%CDIV^MATH 49

$%CEXP^MATH 49, 128, 131

$%CLOG^MATH 49

$%CMUL^MATH 49, 128, 131

$%COLLATE^CHARACTER 46

$%COMPARE^CHARACTER 46

$%COMPLEX^MATH 49

$%CONJUG^MATH 49

$%COS^MATH 49, 50, 129, 130

$%COSH^MATH 50, 130, 137, 138

$%COT^MATH 50

$%COTH^MATH 50

$%CPOWER^MATH 50

$%CSC^MATH 50

$%CSCH^MATH 50

$%CSIN^MATH 50, 51

$%CSUB^MATH 51, 131

$%DECDMS^MATH 51

$%DEGRAD^MATH 51

$%DMSDEC^MATH 51, 129-131, 136, 137

$%E^MATH 51

$%EXP^MATH 51, 129, 130

$%LOG^MATH 51, 52, 126-130

$%LOG10^MATH 52

$%MTXADD^MATH 52

$%MTXCOF^MATH 52, 133

$%MTXCOPY^MATH 52, 133

$%MTXDET^MATH 52, 133, 134

$%MTXEQU^MATH 52, 135

$%MTXINV^MATH 52, 53

$%MTXMUL^MATH 53

$%MTXSCA^MATH 53

$%MTXSUB^MATH 53

$%MTXTRP^MATH 53

$%MTXUNIT^MATH 53, 135

$%PI^MATH 53, 126-131, 136, 137

$%PRODUCE^STRING 55

$%RADDEG^MATH 54

$%REPLACE^STRING 55

$%SEC^MATH 54

$%SECH^MATH 54

$%SIGN^MATH 54

$%SIN^MATH 54, 129, 130

$%SINH^MATH 54, 130, 131, 138

$%SQRT^MATH 54, 126-128, 130, 137

$%TAN^MATH 54, 55

$%TANH^MATH 55

$%Y 45

$%Z 45

$ASCII 18, 19, 33, 38, 57

$CHAR 33, 34, 59, 102

$DATA 20, 22, 34, 36, 74-76, 78, 83, 98, 132-136, 138

$DEVICE 19, 20, 27, 31, 80, 84, 89, 91, 101, 102, 104

$ECODE 10, 11, 17, 27, 28, 42, 84, 85, 126-128, 130, 132, 137

$ESTACK 11, 28, 78, 79

$ETRAP 10, 11, 28, 78, 79, 84, 86

$EXTRACT 33-35, 39, 40, 57, 85, 119, 138

$FIND 35, 39, 57, 138

$FNUMBER 35, 36, 118

$GET 20, 36, 38, 126-138

$HOROLOG 28

$IO 21, 27-29, 31, 68, 80, 88

$IOREFERENCE 28, 29, 88

$JOB 18, 20-22, 29, 30, 38, 65, 73, 99, 117

$JUSTIFY 37, 85, 131, 138

$KEY 29, 84, 101, 102

$LENGTH 34, 35, 37, 39, 57, 85, 131, 138

$NAME 37, 117

$ORDER 38, 39, 58, 83, 95, 138

$PIECE 39, 40, 84, 119, 128, 131, 138

$PIOREFERENCE 29

$PRINCIPAL 21, 28-30

$QLENGTH 40

$QSUBSCRIPT 40, 84, 86, 119

$QUERY 30, 40, 41, 95

$QUIT 11, 30

$RANDOM 41, 117

$REFERENCE 26, 30, 41, 65, 84

$REVERSE 42

$SELECT 42, 117, 126-132, 136, 137

$STACK 11, 28, 30, 42, 43, 85, 119

$STORAGE 30, 67

$SYSTEM 2, 22, 30, 38, 96

$TEXT 43, 63, 119, 122

$TLEVEL 9-11, 26, 31, 72, 81, 87, 88

$TRANSLATE 44, 126-132, 134, 136-138

$TRESTART 10, 26, 31, 87

$VIEW 33, 44

$X 26, 31, 82, 84, 86, 88-90, 96, 98, 101, 117

$Y 27, 31, 82, 84, 86, 88-90, 96, 98, 101, 117

$Z 22, 31, 32, 44

\*\* 56

^$CHARACTER 7, 17-20, 38, 46, 47, 82

^$DEVICE 17, 19, 20, 80, 91

^$GLOBAL 17, 20, 38, 46, 47, 117

^$JOB 17, 18, 20-22, 38, 65, 73, 117

^$LIBRARY 17, 21

^$LOCK 17, 21

^$ROUTINE 17, 21, 22, 117

^$SYSTEM 17, 22, 38

^$Z 17, 22

Action 35

actual 65, 66, 69, 73, 117

actuallist 25, 26, 61, 65, 69, 73, 118

actualname 65, 66, 118

Addition 95

algoref 18-20, 22, 118

Alpha 79

alternation 59

And operator ( & ) 58

argument 62

Argument

indirection 62

QUIT 66, 80

Arithmetic Binary Operators 56

Arithmetic Operations 95

Array 12-14

ASCII 6, 7, 34, 82, 90, 96, 106

binaryop 55, 56, 118

Blocks 69

BREAK 67

Call-by-reference 65, 66, 73

Call-by-value 65, 73

Case

Lower 17

Upper 17

case sensitivity 6, 26, 32, 59, 60

CB 6, 59, 118

Character set profile 105

charset 7, 18, 19, 21, 22, 34, 38, 46, 47, 59, 91, 105, 106, 108-110, 118, 125

charsetexpr 7, 18-20, 22, 118

charspec 59, 118

CLOSE 68, 80

closeargument 68, 118

Codes

$FNUMBER 35

Pattern matching 59

Collation 125

statement 7-11, 20, 26, 42, 43, 60-64, 66, 67, 69-72, 77-80, 84, 86, 88, 90, 91, 96, 117

Statement argument indirection 6, 62

Statement structure 8, 61, 96, 97

statements 7, 8, 97

statementword 11, 42, 60-62, 67, 90, 118

Commas 27

comment 7-9, 61

Commit 9, 10, 87

Complex interpretation 45

Complex Numbers 56

Concatenation operator ( \_ ) 56

Conditional statements

ELSE 70

FOR 70

IF 72

post 62

Contains operator ( [ ) 57

CONTEXT-STRUCTURE 9, 10

Control 7

Control-sequences 82

controlmnemonic 66, 67, 79, 80, 89, 98, 99, 101, 102, 104, 117, 118

CR 6-8, 89, 106, 110, 118

cs 8, 118

Current Device 27, 28, 68, 80-82, 88, 89

Data type 12

Data values

Numeric 94

DATA-CELL 3, 14, 15, 34, 65, 66, 74, 75, 79

Defining occurrence 8, 63

Descendants 13, 34, 74, 75, 78

device 19, 29, 30, 98

Current 27, 89

Principal 28, 29

deviceattribute 19, 67, 68, 80, 118

devicestatement 67, 118

devicekeyword 67, 68, 80, 118

deviceparam 20, 67, 68, 91, 118

deviceparameter 67, 68

Deviceparameters 67, 68, 79, 80, 88, 118

devicexpr 19-21, 118

devn 68, 79, 88

Difference operator ( - ) 56

digit 6, 7, 23, 79, 89

Division operator (/) 57

dlabel 43, 63, 64, 118

DO 9, 13, 61, 63-65, 68, 72, 80, 90

doargument 9, 16, 61, 66, 69, 70, 80, 118

ecode 8, 10, 11, 17, 22, 23, 26-28, 36, 37, 41-43, 47-49, 52, 54, 56, 60, 61, 63-73, 77, 78, 80-83, 85-87, 91, 94, 96, 118, 126-128, 130, 132, 137

M1 17, 117

M10 60, 117

M100 117

M11 61, 117

M12 63, 117

M13 63-65, 117

M14 69, 117

M15 70, 71, 117

M16 71, 80, 117

M17 81, 117

M18 82, 117

M19 78, 117

M2 36, 117

M20 26, 117

M21 8, 117

M22 117

M23 117

M24 117

M26 23, 64, 68, 73, 117

M27 10, 117

M28 47, 48, 52, 54, 117, 126-128, 130, 132, 137

M29 117

M3 41, 117

M30 117

M31 117

M32 67, 117

M33 117

M35 80, 117

M36 80, 117

M37 117

M38 117

M39 37, 117

M4 42, 117

M40 66, 73, 117

M41 77, 117

M42 81, 117

M43 86, 117

M44 87, 117

M45 72, 117

M5 43, 117

M56 94, 117

M57 8, 117

M58 26, 117

M6 56, 117

M7 56, 117

M75 96, 117

M8 56, 117

M9 49, 56, 117

M90 117

M100 91

M13 65

M28 47, 48, 52, 54

M75 96

M9 49, 56

M90 86

ELSE 9, 70, 72

Embedded programs 11

Embedded;SQL 120

Empty string 12, 15, 19, 23, 27-31, 36, 38, 41, 44, 57, 86, 94

Empty value 68

emptystring 18, 19, 118

Endless loops 71

entryref 43, 63, 64, 69, 71, 73, 118

environment 4, 11, 12, 14, 16, 17, 22, 23, 29, 32, 37, 40, 41, 43, 64, 66, 68, 69, 71, 73, 76, 80, 86, 95, 97, 99, 117, 118, 123

eoffset 43, 118

eol 7, 8, 11, 43, 61, 70, 90, 96, 118, 122

eor 7, 9, 66, 69, 80, 118

Equality 24, 57

Equals operator ( = ) 57

Erroneous 37

Error 8, 95

Errors 36, 56, 64

Evaluation 6

statement argument 6, 62

expression 6, 55, 90

indirection 6, 61

naked indicator 17

parameters 65

exampleargument 6, 118

examplestatement 6, 118

Execution 4, 9, 20, 90, 91

Execution level 9, 26, 69, 72, 80

exfunc 9, 15, 23, 25, 26, 30, 42, 61, 64-66, 69, 80, 81, 118

exp 23, 24, 51, 118, 129-132

exponentiation 56, 95

expr 6, 12, 13, 16-23, 25, 28, 32-38, 41, 42, 65-68, 70, 73, 76, 79-90, 99, 101, 118, 119

expratom 6, 12, 13, 16, 17, 23, 32, 43, 55, 56, 58, 61, 62, 64, 65, 68, 69, 72, 73, 76-83, 87-90, 118

Expressions 12, 94, 96, 98

expritem 12, 13, 23, 118

exprtail 12, 55, 118

extblock 8, 9, 11, 61, 118

extcode 11, 12, 118

externalroutinename 64, 94, 118

externref 18, 25, 26, 64, 66, 69, 118

extid 11, 12, 118

Extrinsic

functions 9, 13, 25, 63

variables 9, 13, 26, 63

exvar 9, 15, 23, 26, 30, 42, 61, 64-66, 69, 80, 81, 118

FF 6, 7, 89, 106, 110, 118

fncodatom 35, 36, 117, 118

fncode 35, 36, 118

fncodexpr 35, 36, 118

fncodp 35, 36, 118

fncodt 35, 36, 118

Follows operator ( ] ) 57

Follows or equals operator ( ]= ) 57

Follows or equals operators ( ]= ) 57

FOR 9, 13, 66, 69, 70, 80

formalline 7, 8, 26, 61, 64, 65, 69, 73, 118

formallist 7, 8, 10, 26, 65, 66, 69, 73, 81, 117, 118

format 31, 66, 79, 81, 82, 89, 90, 98, 99

forparameter 70, 71, 118

function 23, 32, 33, 63

functionname 18, 33, 118

Global variable 12-14

Global variables 95

glvn 12, 13, 29, 32, 34, 36-41, 73-75, 77, 78, 81-86, 117, 118

gnamind 16, 118

graphic 7, 9, 11, 23, 27

Greater than operator ( > ) 57

Greater than or equal to operator ( >= ) 57

gvn 13, 16, 17, 20, 23, 30, 38, 40, 41, 56, 83, 117, 118

gvnexpr 20, 118

HALT 72, 81

implicit 81

HANG 72, 99

hangargument 72, 118

ident 6, 7, 18, 59, 79, 89, 91, 106, 118, 125

IF 9, 30, 72

ifargument 72, 118

Indicator

Naked 17, 30, 36, 78, 86

Indirection 4, 61, 70, 96, 97

statement argument 6, 62

name 13, 16, 64, 65, 73, 76

pattern 58

subscript 13, 16

Integer Division ( \ ) 56

Integer division operator ( \ ) 56

Integer interpretation 25

Interpretation

complex 45

integer 25, 56, 95, 96

numeric 25, 32, 56, 57, 95

truth-value 25, 32, 58, 95

intexpr 25, 33-37, 39-43, 63, 64, 81, 82, 84, 86, 89, 90, 118

intlit 8, 23, 24, 43, 59, 60, 63, 97, 118

JIS90 125

JOB 9, 29, 30, 62-64, 73

jobactuallist 118

jobargument 9, 23, 61, 66, 73, 80, 118

jobenv 73, 118

jobparameters 73, 118

Katakana 125

KILL 13-15, 17, 66, 73, 74

implicit 66

killarglist 73-75, 118

killargument 73, 118

KSUBSCRIPTS 74

KVALUE 75, 132, 133, 135, 136

L 6, 8, 13, 16, 17, 21, 27, 33-35, 39, 40, 42, 45, 53, 59, 62, 65, 68-79, 81-83, 87-90, 94, 95, 99, 105-108, 110, 112, 113, 115, 116, 118, 125-138

label 7-9, 26, 43, 63, 64, 90, 97

label offsets 43, 65

labelref 18, 23, 25, 26, 63-66, 69, 73, 118

leftexpr 84, 118

leftrestricted 84, 118

Less than operator ( < ) 57

Less than or equal to operator ( < ) 57

Less than or equal to operator ( <= ) 57

LEVEL vii, 2-4, 7-9, 11, 12, 16, 22, 26, 28, 30, 42, 43, 56, 69, 72, 80, 81, 98, 117-119

execution 9, 69, 72, 80

indicator 7

line 8, 9, 69

precedence 56

LF 6-8, 89, 106, 110, 118

libdatatype 45, 46, 118

definition 45

library 21, 45, 64, 65

libraryelement 21, 44, 45, 64, 65, 118

libraryelementdef 45, 118

definition 45

libraryelementexpr 21, 118

definition 21

libraryexpr 21, 118

definition 21

libraryopt 45, 46, 118

definition 45

libraryparam 45, 46, 118

definition 45

libraryref 25, 26, 64, 65, 118

libraryresult 45, 46, 118

definition 45

line 4, 7-9, 43, 44, 61, 63, 64, 66, 69-73, 83, 96

Line references 9, 63

lineref 63, 117, 119

lname 73-75, 78, 87, 119

lnamind 13, 119

Local variable 12-14, 82

Local variables 94, 98

LOCK 4, 9, 21-23, 30, 62, 72, 76, 77, 88, 117, 119

LOCK-LIST 9, 72, 76, 77

LOCK-UNIVERSE 76

lockargument 76, 77, 119

Lockspace 76

LOCK-LIST 9, 72, 76, 77

Logical operators 58

logicalop 57, 58, 119

Lower Case 17

lvn 13-15, 56, 58, 66, 70, 71, 82, 117, 119

M

charset 106

M Standard Library 44, 46

M1 17, 117

M10 60, 117

M100 117

M11 61, 117

M12 63, 117

M13 63-65, 117

M14 69, 117

M15 70, 71, 117

M16 71, 80, 117

M17 81, 117

M18 82, 117

M19 78, 117

M2 36, 117

M20 26, 117

M21 8, 117

M22 117

M23 117

M24 117

M26 23, 64, 68, 73, 117

M27 10, 117

M28 47, 48, 52, 54, 117, 126-128, 130, 132, 137

M29 117

M3 41, 117

M30 117

M31 117

M32 67, 117

M33 117

M35 80, 117

M36 80, 117

M37 117

M38 117

M39 37, 117

M4 42, 117

M40 66, 73, 117

M41 77, 117

M42 81, 117

M43 86, 117

M44 87, 117

M45 72, 117

M5 43, 117

M56 94, 117

M57 8, 117

M58 26, 117

M6 56, 117

M7 56, 117

M75 96, 117

M8 56, 117

M9 49, 56, 117

M90 117

mant 23, 24, 119

MCODE 43

MERGE 60, 77, 119

mergeargument 77, 119

Metalanguage 6

Minus operator (-) 25, 32

mnemonicspace 19, 29, 66-68, 79, 80, 82, 88-90, 98, 99, 102, 117, 119

mnemonicspacename 79, 119

mnemonicspec 19, 79, 80, 119

mnemonispace 79

Modulo 95

Modulo operator ( # ) 56

Multiplication 95

Naked indicator 16, 17, 30, 36, 41, 66, 67, 78, 83, 85, 86

Naked reference 16, 17, 32, 37

name 6-8, 13-16, 18, 21, 22, 26, 32-34, 45, 47, 64-66, 68, 69, 73-76, 81, 83, 86, 91, 94, 95, 97, 105

Name indirection 13, 64, 65, 73, 76

NAME-TABLE 10, 13, 14, 34, 65, 74, 75

namevalue 4, 30, 32, 37, 40, 41, 46, 86, 99, 117, 119

NAME-TABLE 4, 13-15, 34, 65, 66, 74, 75, 79

Nesting 98

levels 96

NEW 12-15, 28, 66, 78

implicit 66

newargument 78, 119

newsvn 78, 119

noncomma 27, 119

nonquote 23, 119

Not operator (') 32, 55, 57, 58, 60

nref 9, 21, 23, 72, 76, 77, 119

Numeric Data Values 23, 94

Numeric expression evaluation 36

Numeric Interpretation 24, 25

Numeric relations 57

numexpr 25, 35-37, 62, 63, 70-72, 91, 119

numlit 23-25, 29, 59, 94, 95, 119

OB 6, 59

omitted-parameter 65, 66

OPEN 19, 28-30, 62, 79, 88

openargument 79, 80, 119

openparameters 79, 80, 119

Operand 56, 57

operator

metalanguage 6

Operators 56, 57, 96

arithmetic 56

concatenation 56

dual 57

metalanguage 6

pattern match 58

precedence of 56

relational 57

truth-value 32

unary 32

Or operator ( ! ) 58

Order of evaluation 9, 32, 42, 55, 61, 65, 69, 70, 89

Ordering Sequence 38, 58

OUTSTALLED 20, 91

OUTTIMEOUT 20, 91

OVERLAP 77

Ownership

device 80

Ownership of devices 68

packagename 64, 119

Parameter 99

Parameter passing 12, 14, 15, 26, 64-66, 69

Pass-by-value 73

patatom 58-60, 119

patcode 19, 58, 59, 91, 105, 106, 110, 119, 125

patgrp 59

patnonY 59, 119

patnonYZ 59, 119

patnonZ 59, 119

pattern 55, 58

Pattern indirection 58

Pattern match operator (?) 55, 58

place 43

Plus operator (+) 25, 32

Post conditionals 62

postcond 10, 61, 62, 67-69, 71-83, 87-90, 119

PROCESS-STACK 9, 11, 14, 26, 30, 69, 81

processid 20-22, 99, 119

processparameters 21, 73, 119

PROCESS-STACK 5, 10, 14, 15, 26, 66, 69, 73, 78, 81, 88, 98

Product operator ( \* ) 56

Programs

embedded 11

QUIT 9, 11, 26, 28, 30, 66, 69, 71, 80, 81

implicit 9, 69, 80

Quotient operator ( / ) 56

Quotient operator, integer ( \ ) 56

READ 13, 20, 29, 30, 62, 79, 81, 82, 90, 91, 98

readargument 81, 119

readcount 81, 82, 119

relation 57, 58

repcount 58-60, 119

RESTART 9, 10, 31, 81, 87, 88, 119

Transaction 10

Restartable 9

restartargument 10, 87, 88, 119

Results 96

REVERSE 33

rexpratom 13, 16, 17, 119

rgvn 13, 16, 119

RLOAD 82

rlvn 13, 119

Rollback 9, 10, 17, 31, 72, 87, 117

routine 7, 9, 11, 12, 22, 43, 63, 64, 79, 81, 83

size 97

Routine execution 9

routineargument 82, 83

routineattribute 83

routinebody 7-9, 11, 119

routinehead 7, 119, 122

routinekeyword 83

routinename 7, 16, 21, 43, 63, 64, 119

routineparam 82, 83

routineparameters 82

routineref 23, 43, 63, 64, 79, 82, 83, 119

routinexpr 21, 22, 119

RSAVE 81, 83

rssvn 17

Scope

Transaction 9, 77

Scoping 13, 14, 66, 69

FOR 70

variable 78

SERIAL 87, 88

Serializable 9

SET 11, 13, 14, 28, 66, 78, 83

implicit 66

setargument 83, 84, 86, 119

setdestination 83, 86, 119

setev 84-86, 119

setextract 84, 85, 119

setleft 83-86, 119

setpiece 84, 85, 119

setqsub 84, 86, 119

Sorts After 58

Sorts after operator ( ]] ) 57

Sorts after or equals operator ( ]]= ) 57, 58

SP 6-8, 43, 62, 67-73, 76-83, 87-90, 107, 111, 119

Spaces 6, 36, 37, 89

in statements 61

Special variables 26

SQL 11, 120

ssvn 13, 17, 22, 23, 38, 56, 80, 99, 117, 119

ssvname 17

ssvnamind 17

stackcode 42, 43, 119

stackcodexpr 42, 119

strconst 59, 119

String operators 57

String relations 57

Strings 12, 23, 95, 96

strlit 23, 29, 58-60, 81, 82, 119

Structured system variable 17

sublit 32, 94, 95, 119

subnonquote 32, 119

Subscript 4, 12-16, 32, 34, 38, 70, 76, 84, 86

Subscript indirection 13, 16

Subscripts 17, 94

Subtraction 95

Sum operator ( + ) 56

svn 22, 23, 26, 27, 56, 78, 79, 117, 119

Symbol table 13

Syntax 6, 26, 27, 33, 60, 62, 65

system 22

systemexpr 22, 119

TCOMMIT 9, 31, 60, 86, 87

Terminator 29

textarg 43, 119

timeout 19, 20, 62, 63, 67, 73, 76, 77, 79-82, 90, 91, 117

TRANSACTION 9, 10, 12, 31, 77, 87, 88, 99, 117, 119

Transaction Processing 9, 99

TRANSACTIONID 87, 88

transparameters 10, 87, 88, 119

TRESTART 9, 10, 31, 60, 87

TROLLBACK 9, 11, 31, 60, 87

Truth-value 27

Truth-Value Interpretation 25, 32, 42

truthop 55-57, 119

tsparam 87, 88, 119

TSTART 9, 10, 31, 60, 81, 87, 88, 99, 119

tstartargument 10, 87, 88, 119

tstartkeyword 87, 119

tvexpr 25, 42, 62, 69, 72, 90, 119

Unary operators 32

unaryop 23, 32, 119

Undefined 13, 14, 30, 41, 65

Upper case 17

USE 19, 28, 29, 88, 89

useargument 88, 119

V 2, 6, 13, 16-22, 30, 35, 40-43, 58, 62, 64, 65, 68, 69, 72-83, 87-90, 105, 106, 108, 112, 113, 119

VALUE-TABLE 14

Values

arithmetic 95

VALUE-TABLE 3, 5, 14, 15

Variable 12, 34, 36

global 12, 13, 95

local 12, 13, 82, 94, 98

special 96

VB 6, 16, 17, 43, 64, 73, 76, 95, 119

Vertical bar 6

VIEW 88

WRITE 20, 79, 82, 89-91, 98

writeargument 82, 89, 119

xargument 16, 66, 69, 80, 90, 119

XECUTE 9, 43, 61, 62, 90, 97

Z 90