INTERNATIONAL STANDARD

ISO/IEC 11756

Second edition
1999-06-01

Information technology — Programming languages — M

Technologies de l'information — Langages de programmation — M

Reference number
ISO/IEC 11756:1999(E)

Contents

1. Scope 1

2. Normative references 1

3. Conformance 2
3.1 Implementations 2
3.2 Programs 2

4. Definitions 4

5. Metalanguage description 7

6. Routine routine 9
6.1 Routine head routinehead 9
6.2 Routine body routinebody 10
6.2.1 Level line levelline 10
6.2.2 Formal line formalline 10
6.2.3 Label label 10
6.2.4 Label separator ls 11
6.2.5 Line body linebody 11
6.3 Routine execution 11
6.3.1 Transaction processing 12
6.3.2 Error processing 13
6.4 Embedded programs 14

7. Expression expr 15
7.1 Expression atom expratom 15
7.1.1 Variables 15
7.1.2 Variable name glvn 16
7.1.2.1 Local variable name lvn 16
7.1.2.2 Local variable handling 16
7.1.2.3 Process-stack 19
7.1.2.4 Global variable name gvn 20
7.1.3 Structured system variable ssvn 21
7.1.3.1 ^$CHARACTER 21
7.1.3.2 ^$DEVICE 23
7.1.3.3 ^$GLOBAL 23
7.1.3.4 ^$JOB 24
7.1.3.5 ^$LOCK 24
7.1.3.6 ^$ROUTINE 24
7.1.3.7 ^$SYSTEM 25
7.1.3.8 ^$Z[unspecified] 25
7.1.3.9 ssvns specifying default environments 25
7.1.4 Expression item expritem 26
7.1.4.1 String literal strlit 26

7.1.4.2 Numeric literal numlit . . . 27
7.1.4.3 Numeric data values . . . 27
7.1.4.4 Meaning of numlit . . . 28
7.1.4.5 Numeric interpretation of data . . . 28
7.1.4.6 Integer interpretation . . . 29
7.1.4.7 Truth-value interpretation . . . 29
7.1.4.8 Extrinsic function exfunc . . . 29
7.1.4.9 Extrinsic special variable exvar . . . 30
7.1.4.10 Intrinsic special variable names svn . . . 30
7.1.4.11 Unary operator unaryop . . . 34
7.1.4.12 Name value namevalue . . . 35
7.1.5 Intrinsic function function . . . 35
7.1.5.1 $ASCII . . . 36
7.1.5.2 $CHAR . . . 37
7.1.5.3 $DATA . . . 37
7.1.5.4 $EXTRACT . . . 37
7.1.5.5 $FIND . . . 38
7.1.5.6 $FNUMBER . . . 38
7.1.5.7 $GET . . . 39
7.1.5.8 $JUSTIFY . . . 40
7.1.5.9 $LENGTH . . . 40
7.1.5.10 $NAME . . . 40
7.1.5.11 $ORDER . . . 41
7.1.5.12 $PIECE . . . 42
7.1.5.13 $QLENGTH . . . 43
7.1.5.14 $QSUBSCRIPT . . . 44
7.1.5.15 $QUERY . . . 44
7.1.5.16 $RANDOM . . . 45
7.1.5.17 $REVERSE . . . 45
7.1.5.18 $SELECT . . . 45
7.1.5.19 $STACK . . . 45
7.1.5.20 $TEXT . . . 47
7.1.5.21 $TRANSLATE . . . 47
7.1.5.22 $VIEW . . . 48
7.1.5.23 $Z . . . 48
7.2 Expression tail exprtail . . . 48
7.2.1 Binary operator binaryop . . . 48
7.2.1.1 Concatenation operator . . . 49
7.2.1.2 Arithmetic binary operators . . . 49
7.2.2 Truth operator truthop . . . 49
7.2.2.1 Relational operator relation . . . 49
7.2.2.2 Numeric relations . . . 50
7.2.2.3 String relations . . . 50
7.2.2.4 Logical operator logicalop . . . 50
7.2.3 Pattern match pattern . . . 51

8. Commands . . . 52
8.1 General command rules . . . 52
8.1.1 Spaces in commands . . . 53
8.1.2 Comment comment . . . 54
8.1.3 Command argument indirection . . . 54
8.1.4 Post conditional postcond . . . 54
8.1.5 Command timeout timeout . . . 55
8.1.6 Line reference lineref . . . 55
8.1.6.1 Entry reference entryref . . . 55
8.1.6.2 Label reference labelref . . . 56
8.1.6.3 External reference externref . . . 57

8.1.7 Parameter passing 57
8.2 Command definitions 59
8.2.1 BREAK 59
8.2.2 CLOSE 59
8.2.3 DO 60
8.2.4 ELSE 61
8.2.5 FOR 61
8.2.6 GOTO 63
8.2.7 HALT 63
8.2.8 HANG 63
8.2.9 IF 64
8.2.10 JOB 64
8.2.11 KILL 65
8.2.12 LOCK 66
8.2.13 MERGE 67
8.2.14 NEW 68
8.2.15 OPEN 69
8.2.16 QUIT 70
8.2.17 READ 71
8.2.18 SET 73
8.2.19 TCOMMIT 75
8.2.20 TRESTART 76
8.2.21 TROLLBACK 76
8.2.22 TSTART 76
8.2.23 USE 77
8.2.24 VIEW 78
8.2.25 WRITE 78
8.2.26 XECUTE 79
8.2.27 Z 80

9. Character Set Profile charset 80

10. Character set 81

11. Expression elements 81
11.1 Names 81
11.2 External routines and names 81
11.3 Local variables 81
11.3.1 Number of local variables 81
11.3.2 Number of subscripts 81
11.3.3 Values of subscripts 82
11.4 Global variables 82
11.4.1 Number of global variables 82
11.4.2 Number of subscripts 82
11.4.3 Values of subscripts 82
11.4.4 Number of nodes 82
11.5 Data types 82
11.6 Number range 83
11.7 Integers 83
11.8 Character strings 83
11.9 Special variables 83

12. Expressions 83
12.1 Nesting of expressions 83
12.2 Results 83
12.3 External references 84

13 Routines and command lines 84
13.1 Command lines 84
13.2 Number of command lines 84
13.3 Number of commands 84
13.4 Labels 84
13.5 Number of labels 84
13.6 Number of routines 84

14 External routine calls 85

15 Character Set Profiles 85

16 Indirection 85

17 Storage space restrictions 85

18 Process-stack 86

19 Formats 86
19.1 mnemonicspace 86
19.2 controlmnemonic 86
19.3 Parameters 87

20 Transaction processing 87
20.1 Number of modifications in a TRANSACTION 87
20.2 Number of nested TSTARTs within a TRANSACTION 87

21 Other portability requirements 87

22 The binding 88
22.1 Control-functions with an effect on $X or $Y or both 88
22.2 Control-functions with an effect on $KEY 89
22.3 Control-functions with an effect on $DEVICE 89
22.4 Open-ended definitions 90

23 Portability issues 91
23.1 Implementation 91
23.2 Application 91

24 Conformance 92

Annex A 93

Annex B 96

Annex C 97

Annex D 99

Annex E 101

Annex F 103

Annex G 105

Annex H 106

Index 109

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

International Standard ISO/IEC 11756 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 22, *Programming languages, their environments and system software interfaces*.

This second edition cancels and replaces the first edition (ISO/IEC 11756:1992), which has been technically revised.

Annex A forms an integral part of this International Standard. Annexes B to H are for information only.

Introduction

Section 1 consists of nine clauses that describe the MUMPS 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 MUMPS actually be interpreted). The dynamic syntax takes into account transfers of control and values produced by indirection.

Clauses 10 through 21 highlight, for the benefit of implementors and application programmers, aspects of the language that must be accorded special attention if M program transferability (i.e., portability of source code between various M 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 M 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 command line length restriction places an implicit limit on the length of any construct which must be expressed entirely within a single command line.

Clauses 22 through 24 describe the binding between M and ANSI X3.64. 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 M 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 M implementation and the device is left to the implementor.

This page intentionally left blank

Information technology- Programming languages - M

1. Scope

This International Standard describes the M programming language.

2. Normative references

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

ISO/IEC 9075:1992, Information technology -- Database languages -- SQL
ANSI X3.4-1990, (ASCII Character Set)
ANSI X3.64-1979, R1990 (ANSI Terminal Device Control Mnemonics)

3. Conformance

3.1 Implementations

A *conforming implementation* shall

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

b) reject all code that contains errors, where such error detection is required by the International 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 X11.1-*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 International 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 International 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 X11.1-*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 International Standard, shall not exceed the limits and restrictions specified in Section 2 of the International 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 International Standard, the following definitions apply.

4.1 argument (of a command): M command words are verbs. Their arguments are the objects on which they act.
4.2 array: M 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: One or more lines of code within a routine that execute in line as a unit. The argumentless DO command introduces a block, and each of its lines begins with one or more periods. Blocks may be nested. See level.
4.5 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.
4.6 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.
4.7 call: A procedural process of transferring execution control to a **callee** by a **caller**.
4.8 callee: The recipient of a **call**.
4.9 caller: The originator of a **call**.
4.10 command: A command word (a verb), an optional conditional expression, and zero or more arguments. Commands initiate all actions in M.
4.11 computationally equivalent: The result of a procedure is the same as if the code provided were executed by a M program without error. However, there is no implication that executing the code provided is the method by which the result is achieved.
4.12 concatenation: The act or result of joining two strings together to make one string.
4.13 conditional expression: Guards a command (sometimes an argument of a command). Only if the expression's value is true does the command execute (on the argument). See truthvalue.
4.14 contains: a logical operator that tests whether one string is a substring of another.
4.15 data-cell: in the formal model of M 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.
4.16 descriptor: uniquely defines an element. It comprises various characteristics of the element that distinguish the element from all other similar elements.
4.17 device-dependent: That which depends on the device in question.
4.18 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.
4.19 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.
4.20 evaluate: to derive a value.
4.21 execute: to perform the operations specified by the commands of the language.
4.22 extract: to retrieve part of a value, typically contiguous characters from a string.
4.23 extrinsic: a function or variable defined and created by M code, distinct from the primitive functions or special variables of the language. See intrinsic.
4.24 follow: to come after according to some ordering sequence. See also sorts after.
4.25 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.
4.26 global variable: a scalar or array variable that is public, available to more than one job,

and persistent, outliving the job. See local variable.
4.27 graphic: a visible character (as opposed to most control characters).
4.28 hidden: unseen. The NEW command hides local variables. Also pertains to unseen elements invoked to define the operation of some commands and functions.
4.29 intrinsic: a primitive function or variable defined by the language standard as opposed to one defined by M code. See extrinsic.
4.30 job: A single operating system process running a M program.
4.31 label: Identifies a line of code.
4.32 level: The depth of nesting of a block of code lines. The first line of a routine is at level 1 and successively nested blocks are at levels 2, 3, . . . Formally, the level of a line is one plus li. Visually, li periods follow the label (if any) and precede the body of the line. See block.
4.33 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.
4.34 lock: To claim or obtain exclusive access to a resource.
4.35 mapping: The logical association or substitution of one element for another.
4.36 map: The act of mapping.
4.37 metalanguage: Underlined terms used in the formal description of the M language.
4.38 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. M explicitly defines the result of this computation.
4.39 multidimensional: Used in reference to arrays to indicate that the array can have more than one dimension.
4.40 naked: A shorthand reference to one level of the tree forming a global array variable. The full reference is defined dynamically.
4.41 name-table: In the formal model of M execution, a set of variable names and their pointers to data-cells.
4.42 node: One element of the tree forming an array. It may have a value and it may have descendants.
4.43 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).
4.44 null string: 1. A string consisting of 1 (one) NULL character; 2. A string consisting of 0 (zero) characters.
4.45 object: An entity considered as a whole in relation to other entities.
4.46 own: To have exclusive access to a resource. In M this pertains to devices.
4.47 parameter: A qualifier of a command 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.
4.48 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.
4.49 parameter passing: This alliterative phrase refers to the association of actual parameters with formal parameters when calling a subroutine or function.
4.50 partition: The random access memory in which a job runs.
4.51 piece: A part of a string, a sub-string delimited by chosen characters.
4.52 pointer: Indirection allows one M variable to refer, or point to, another variable or the argument of a command.
4.53 portable: M code that conforms to the portability section of the International Standard.
4.54 post-conditional: See conditional expression.
4.55 primitives: The basic elements of the language.
4.56 process-stack: In the formal model of M execution, a push-down stack that controls the execution flow and scope of variables.
4.57 relational: Pertaining to operators that compare the values of their operands.
4.58 scalar: Single-valued, without descendants. See array.
4.59 scope (of a command)**:** The range of other commands affected by the command, as in loop control, block structure, and

conditional execution.

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

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

4.62 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.

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

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

4.65 type: M 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.

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

4.67 undefined: Pertaining to a variable that is not visible to a command.

4.68 unsubscripted: See scalar.

4.69 value-denoting: Representing or having a value.

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

4.71 variable: M 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

The following visible representations of ASCII characters required in the defined syntactic objects are used: SP (space), CR (carriage-return), LF (line-feed), and FF (form-feed).

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 name [, name] ... .

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 command with its argument list by

```
examplecommand   ::=   EXAMPLE SP L exampleargument
```

where

```
                      | expr                               |
exampleargument  ::=  |                                    |
                      | @ expratom V L exampleargument     |
```

This example states: after evaluation of indirection, the command 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 command 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.

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

See clause 9 for a definition of charset.

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

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 |
                                  └       ┘

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")
```

See 7.1.3.1 for definitions of ^$CHARACTER and charsetexpr.

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

eol            ::=       CR LF
```

6.2 Routine body routinebody

The routinebody is a sequence of lines terminated by an eor. Each line starts with one ls which may be preceded by an optional label and formallist. The ls is followed by zero or more li (level-indicator) which are followed by zero or more commands and a terminating eol. If there is a comment it is separated from the last command of a line by one or more spaces.

```
routinebody ::=  line ... eor

line        ::=  | levelline  |
                 | formalline |

eor         ::=  CR FF
```

6.2.1 Level line levelline

A levelline is a line that does not contain a formallist. A levelline may have a LEVEL greater than one. The LEVEL of a line is the number plus one of li. Subclause 6.3 (Routine Execution) describes the effect a line's LEVEL has on execution.

```
levelline   ::=  [ label ]  ls  [ li ] ... linebody

li          ::=  . [ SP ] ...
```

6.2.2 Formal line formalline

A formalline contains both a label and a formallist which is a (possibly empty) list of variable names. These names may contain data passed to this subroutine (see 8.1.7 Parameter passing). A formallist shall only be present on a line whose LEVEL is one, i.e., does not contain an li.

```
formalline  ::=  label formallist ls linebody

formallist  ::=  ( [ L name ] )
```

If any name is present more than once in the same formallist an error condition occurs with ecode="M21".

6.2.3 Label label

Each occurrence of a label to the left of ls in a line 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 |
```

See 7.1.4.2 for a definition of intlit.

6.2.4 Label separator ls

A label separator (ls) precedes the linebody of each line. A ls consists of one or more spaces. The flexible number of spaces allows programmers to enhance the readability of their programs.

```
ls          ::=  SP ...
```

6.2.5 Line body linebody

The linebody consists of an optional sequence of commands and an optional comment. Note that the comment always comes after any commands in the line (see 8.1.2 for more about comments). Individual commands are separated by one or more spaces (see 8.1.1 for more about spaces in commands). The end of the line is terminated by a CR LF character sequence.

```
                 ┌                                    ┐
                 │ commands [ cs comment ]            │
linebody    ::=  │ [ commands cs ] extsyntax          │ eol
                 │         comment                    │
                 └                                    ┘

commands    ::=  command [ cs command ] ...
```

See clause 8 for a definition of command.

```
cs          ::=  SP ...

comment     ::=  ; [ graphic ] ...
```

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

6.3 Routine execution

Routines are executed in a sequence of blocks. Each block is dynamically defined and is invoked by the instance of an argumentless DO command, a doargument, an exfunc, or an exvar. Each block consists of a set of lines that all have the same LEVEL; the block begins with the line reference implied by the DO, exfunc, or exvar and ends with an implicit or explicit QUIT command. If no label is specified in the doargument, exfunc, or exvar, the first line of the routinebody is used. The *execution level* is defined as the LEVEL of the line currently being executed. Lines which have a LEVEL greater than the current execution level are ignored, i.e., not executed. An implicit QUIT command is executed when a line with a LEVEL less than the current execution level or the eor is encountered, thus terminating this block (see 8.2.16 for a description of the actions of QUIT). The initial LEVEL for a process is one. The argumentless DO command increases the execution level by one. (See also the DO command and GOTO command).

Within a given block execution proceeds sequentially from line to line in top to bottom order. Within a line, execution begins at the leftmost command and proceeds left to right from command to command. Routine flow commands DO, ELSE, FOR, GOTO, IF, QUIT, TRESTART, XECUTE, exfunc and exvar extrinsic functions and special variables, provide exception to this execution flow. (See also 6.3.2 Error Processing.) In general, each command's argument is evaluated in a left-to-right order, except as explicitly noted elsewhere in this document.

6.3.1 Transaction processing

A TRANSACTION is the execution of a sequence of commands 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 8.2.22 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 command 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. M errors do not cause an implicit ROLLBACK. (See the LOCK command for definitions of nref and LOCK-LIST.)

Global variable modifications carried out by commands 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 command 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 8.2.22 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 command (see 8.2.16).

2) the frame is removed and step a repeats.

b) $TEST and the naked indicator are 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.

6.3.2 Error processing

Error trapping provides a mechanism by which a process can execute specifiable commands 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 command.

$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 consists of an implicit GOTO (without changing the PROCESS-STACK) to the following two lines of code where x is the value of $ETRAP. These lines are implicitly incorporated into the current execution environment immediately preceding the next command in the normal execution sequence.

```
ls x eol
ls QUIT:$QUIT "" QUIT  eol
```

For purposes of this transfer each command argument is considered to have its own commandword (see 8.1 General command rules)

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 command.

b) $ECODE is not the empty string and a QUIT command 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 line of code; this line is implicitly incorporated into the current execution environment immediately preceding the next command in the normal execution sequence:

```
ls TROLLBACK:$TLEVEL  QUIT:$QUIT "" QUIT  eol
```

6.4 Embedded programs

An embedded *xxx* M program is a program which consists of M text and text written to the specifications of the *xxx* programming language or standard. Although it is not a routine, an embedded M program conforms to the syntax of a M routinebody.

```
extsyntax  ::=  & extid ( exttext )

exttext    ::=  graphic ... [eol & ls graphic ... ] ...

extid      ::=  | SQL |
```

In exttext each eol & ls sequence is either ignored or, if required by the other programming language or standard, replaced by one or more graphic characters. Exttext is then treated as if the graphic characters following the ls were part of the previous line (a continuation line).

The exact syntax of the remainder of exttext 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).

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

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 M International 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 M environment.

A M 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. These shared variables will normally be stored on secondary peripheral devices such as disks. At the termination of the process, the process-specific variables cease to exist. The variables created for long term (shared) use remain on auxiliary storage devices where they may be accessed by subsequent 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. However, they are generally available to all routines executed within the process's lifetime. M does include certain constructs, the NEW command and parameter passing, which 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     |
                      | 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 commands 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 command, 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 command 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 command 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 command, 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 M 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 command 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.

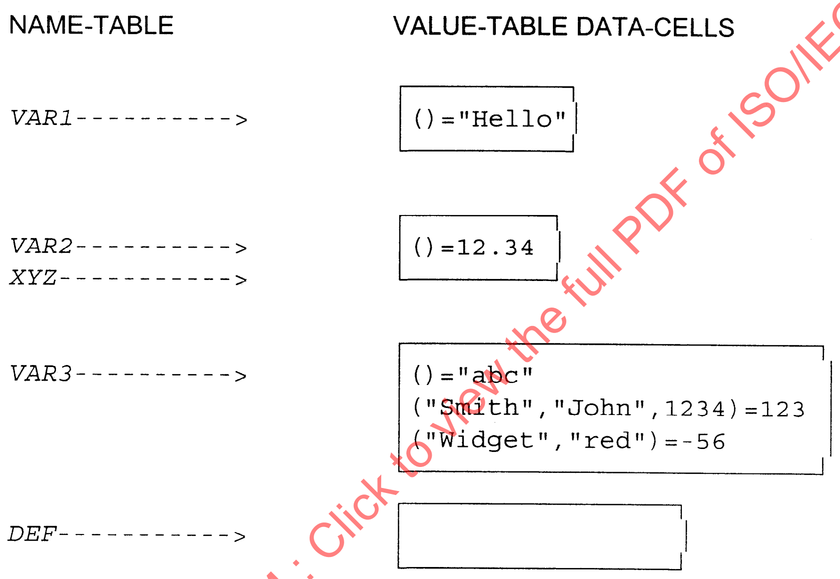
The NAME-TABLE entries contain every non-subscripted variable or array name (name) known, or accessible, by the process in the current environment. The VALUE-TABLE DATA-CELLs contain the set of tuples that represent all variables currently having data-values for the process. Every name (entry) in the 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*.



The initial state of a process prior to execution of any M 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 command.

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 M. It is used as an aid in describing how M appears to work and does not imply that an implementation is required to use such a stack to achieve the specified behavior. Three types of items, or frames, will be placed on the PROCESS-STACK, DO frames (including XECUTEs), extrinsic frames (including exfunc and exvar) and error frames (for errors that occur during error processing):

a) DO frames contain the execution level and the execution location of the doargument or xargument. In the case of the argumentless DO, the execution level, the execution location of the DO command and a saved value of $TEST are saved. The execution location of a process is a descriptor of the location of the command and possible argument currently being executed. This descriptor includes, at minimum, the routinename and the character position following the current command or argument.

b) Extrinsic frames contain saved values of $TEST, the execution level, and the execution location.

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         ::=       | ^ [ | environment | ] 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

$$N(v_1, v_2, \ldots, v_m)$$

then the *m*-tuple $N, v_1, v_2, \ldots, v_{m-1}$, is placed into the naked indicator when the gvn reference is made. A subsequent naked reference of the form

$$\text{^}(s_1, s_2, \ldots, s_i) \qquad (i \text{ positive})$$

results in a global reference of the form

$$N(v_1, v_2, \ldots, v_{m-1}, s_1, s_2, \ldots, s_i)$$

after which the $m+i-1$-tuple $N, v_1, v_2, \ldots, s_{i-1}$ 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 command 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

Structured system variables are denoted by the prefix ^$ followed by one of a designated list of names, followed by a parenthesized list of exprs. These exprs will be called subscripts. Structured system variable names differing only in the use of corresponding upper and lower case letters are equivalent.

The formal definition of ssvn is a choice from among all of the individual ssvn definitions below:

```
           | syntax of ^$CHARACTER structured system variable        |
           | syntax of ^$DEVICE structured system variable           |
           | syntax of ^$GLOBAL structured system variable           |
ssvn  ::=  | syntax of ^$JOB structured system variable              |
           | syntax of ^$LOCK structured system variable             |
           | syntax of ^$ROUTINE structured system variable          |
           | syntax of ^$SYSTEM structured system variable           |
           | syntax of ^$Z[unspecified] structured system variable   |
```

Values may not be assigned to ssvns and ssvns may not be KILLed unless the semantics of these operations are explicitly defined.

The following structured system variables are specified.

7.1.3.1 ^$CHARACTER

^$C[HARACTER] (charsetexpr)

```
        charsetexpr ::= expr V charset
```

See clause 9 (Character Set Profiles) for a definition of 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 exists, ^$CHARACTER(charset) has a value; all nonempty string values are reserved for future extension of the International Standard.

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

Input-Transformation:

^$CHARACTER($charsetexpr_1$, expr V "INPUT" , $charsetexpr_2$) = expr V algoref

```
                        | emptystring     |
                        | $$ labelref     |
algoref        ::=      | $& externref    |
                        | $ functionname  |

emptystring ::=  a string of zero length.
```

See 8.1.6.2 for a definition of labelref. See 8.1.6.3 for a definition of externref. See 7.1.5 for a definition of functionname.

This node specifies the input-transformation algorithm which is performed on a string in the process Character Set Profile charset$_1$ when it is retrieved from a global or routine which uses charset$_2$ or transmitted from a device using charset$_2$. 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:

^$CHARACTER(charsetexpr$_1$, expr V "OUTPUT" , charsetexpr$_2$) = expr V algoref

This node specifies the output-transformation algorithm which is performed on a string in the process Character Set Profile charset$_1$ when it is stored in a global or routine which uses charset$_2$ or transmitted to a device using charset$_2$. 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 M 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.

7.1.3.3 ^$GLOBAL

^$G[LOBAL] (gvnexpr)

```
gvnexpr     ::=  expr V gvn
```

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

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

^$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 command, when a process is created ^$JOB($JOB,"CHARACTER") is set to the charset of the process that created it.

7.1.3.5 ^$LOCK

^$L[OCK] (expr V nref)
will provide information on the existence and operational characteristics of locked names.

See 8.2.12 for a definition of nref.

7.1.3.6 ^$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, ^$ROUTINE(routinexpr) has a value; all nonempty string values are reserved for future enhancement of the International Standard. Process characteristics are stored beneath the ^$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").

7.1.3.7 ^$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 International 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, such as global names, routine names, environment names, and all subscripts of ssvns.

7.1.3.8 ^$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 International Standard without altering the execution of existing programs which observe the rules of the International Standard.

7.1.3.9 ssvns specifying default environments

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

^$JOB(processid,"GLOBAL")	default global environment
^$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 command begins with the routine environment specified in the JOB command, if any. If the command did not specify one, then the initiated process inherits the default routine environment of the initiating process.

Explicit qualification of a labelref, routineref, gvn, or nref 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, or lock 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 )         |
```

See 7.1.5 for a definition of function.

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      ::= $|             | actuallist
                 | externref   |
```

See 8.1.6.2 for a definition of labelref. See 8.1.7 for a definition of actuallist. See 8.1.6.3 for a definition of externref.

Extrinsic functions invoke a subroutine to return a value. When an extrinsic function is executed, the current value of $TEST, the current execution level, and the current execution location are saved in an exfunc 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 command of the formalline specified by the labelref. This formalline 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". Execution of an exfunc to a levelline causes an error condition with ecode="M20".

Upon return from the subroutine the value of $TEST and the execution level are restored, and the value of the argument of the QUIT command that terminated the subroutine is returned as the value of the exfunc.

7.1.4.9 Extrinsic special variable exvar

```
                      | $ labelref |
exvar       ::= $     |            |
                      | externref  |
```

See 8.1.6.2 for a definition of labelref. See 8.1.6.3 for a definition of 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 International Standard contains the following intrinsic special variable names:

```
D[EVICE]
EC[ODE]
ES[TACK]
ET[RAP]
H[OROLOG]
I[O]
J[OB]
K[EY]
P[RINCIPAL]
Q[UIT]
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 International 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.

Syntax	Definition
$D[EVICE]	$DEVICE 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). $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.
$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 command. 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 commands are processed, the value of $ECODE is the empty string. The syntax of a non-empty value returned by $ECODE is as follows: , L ecode ,

```
                | M |
ecode      ::=  | U |  [ noncomma ... ]
                | Z |

noncomma  ::=      any of the characters in graphic except
                   the comma character.
```

Note: <u>ecode</u>s 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.

$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 commands are processed, the value of $ESTACK is 0 (zero).
$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 commands are processed, the value of $ETRAP is the empty string. The value of $ETRAP may be stacked with the NEW command; 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 command. Changing the value of $ETRAP with a SET command 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.
$H[OROLOG]	$HOROLOG 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.
$I[O]	$IO identifies the current I/O device (see 8.2.2 and 8.2.23).
$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.
$K[EY]	$KEY contains the control-sequence which terminated the last READ

	command from the current device (including any introducing and terminating characters). If no READ command 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.
	If a Character Set Profile input-transform is in effect, then this is also applied to the value stored in $KEY.
	See (READ command) and (WRITE command).
$P[RINCIPAL]	$PRINCIPAL identifies the principal I/O device.
	The principal I/O device is the device that is the current device at the moment when a process is started, so that the value of $PRINCIPAL will be equal to the initial value of $IO.
	(See CLOSE and USE commands).
$Q[UIT]	$QUIT 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 commands are processed, the value of $QUIT is 0 (zero).
$ST[ACK]	$STACK 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 commands are processed, the value of $STACK is 0 (zero). See 7.1.2.3 (process-stack) for a description of stack behavior.
$S[TORAGE]	Each implementation must return for the value of $STORAGE an integer which is the number of characters of free space available for use. The method of arriving at the value of $STORAGE is not part of the International Standard.
$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.
$T[EST]	$TEST contains the truth value computed from the execution of an IF command containing an argument, or an OPEN, LOCK, JOB, or READ command with a timeout (see 7.1.4.8, 7.1.4.9, and 8.2.3).
$TL[EVEL]	$TLEVEL indicates whether a TRANSACTION is currently in progress. It is initialized to zero when a process begins. TSTART adds 1 to $TLEVEL. When $TLEVEL is greater than zero, TCOMMIT subtracts 1 from $TLEVEL. A ROLLBACK or RESTART sets $TLEVEL to zero.
$TR[ESTART]	$TRESTART indicates how many RESTARTs have occurred since the initiation of a TRANSACTION. It is initialized to zero when a process begins, and set to zero by the successful completion of TCOMMIT or TROLLBACK. Each RESTART adds 1 to $TRESTART.

$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. 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 command) 8.2.23 (USE command) and 8.2.25 (WRITE command).
$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. 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 command) 8.2.23 (USE command) and 8.2.25 (WRITE command).
$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 International Standard without altering the execution of existing routines which observe the rules of the International 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 as defined in 7.1.4.3 appears as a numlit, spelled as its numeric interpretation.

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 International 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 mth 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 , intexpr_1 , intexpr_2)

Let n be the integer value of intexpr_2. $EXTRACT($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

$F[IND] (expr$_1$, expr$_2$)

This form searches for the leftmost occurrence of the value of expr$_2$ in the value of expr$_1$. If none is found, $FIND returns zero. If one is found, the value returned is the integer representing the number of the character position immediately to the right of the rightmost character of the found occurrence of expr$_2$ in expr$_1$. In particular, if the value of expr$_2$ is empty, $FIND returns 1.

$F[IND] (expr$_1$, expr$_2$, intexpr)

Let *a* be the value of expr$_1$, let *b* be the value of expr$_2$, 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 |
```

<u>fncodt</u>	::=	T t

This form returns a value which is an edited form of <u>numexpr</u>. Each <u>fncodatom</u> is applied to <u>numexpr</u> in formatting the results by the following rules (order of processing is not significant):

<u>fncodatom</u>	<u>Action</u>
<u>fncodp</u>	Represent negative <u>numexpr</u> values in parentheses. Let *A* be the absolute value of <u>numexpr</u>. Use of <u>fncodp</u> will result in the following: 1) If <u>numexpr</u> < 0, the result will be "("_*A*_")". 2) If <u>numexpr</u> '< 0, the result will be " "_*A*_" ".
<u>fncodt</u>	Represent <u>numexpr</u> 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 "-" <u>fncodatom</u>), use of <u>fncodt</u> will result in a trailing space character.
,	Insert comma delimiters every third position to the left of the decimal (present or assumed) within <u>numexpr</u>. Note: no comma shall be inserted which would result in a leading comma character.
+	Force a plus sign ("+") on positive values of <u>numexpr</u>. Position of the "+" (leading or trailing) is dependent on whether or not <u>fncodt</u> is present.
-	Suppress the negative sign "-" on negative values of <u>numexpr</u>.

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

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

$FN[UMBER] (<u>numexpr</u> , <u>fncodexpr</u> , <u>intexpr</u>)

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

7.1.5.7 $GET

$G[ET] (<u>glvn</u>)

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

a) $D(<u>glvn</u>)#10 = 1
The value returned is the value of the variable specified by <u>glvn</u>.

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 expr$_1$, where $S(x)$ is a string of *x* spaces.

$J[USTIFY] (numexpr , intexpr$_1$, intexpr$_2$)

This form returns an edited form of the number numexpr. Let *r* be the value of numexpr after rounding to intexpr$_2$ fraction digits, including possible trailing zeros. (If intexpr$_2$ is the value 0, *r* contains no decimal point.) The value returned is $JUSTIFY(*r*, intexpr$_1$). Note that if $-1 <$ numexpr < 1, the result of $JUSTIFY does have a zero to the left of the decimal point. Negative values of intexpr$_2$ 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.

$L[ENGTH] (expr$_1$, expr$_2$)

This form returns the number plus one of nonoverlapping occurrences of expr$_2$ in expr$_1$. If the value of expr$_2$ 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(s_1, s_2, ..., 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 $NA(Name($s_1$, s_2, ..., 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 a gvn with name *global* then the value of $GET(^$GLOBAL("*global*","COLLATE")) determines the algorithm.

b) If $ORDER does not refer to a gvn, then the value of $GET(^$CHARACTER(^$JOB($JOB,"CHARACTER"),"COLLATE")) determines the algorithm.

c) 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(s_1, s_2, ..., s_n) where s_n may be the empty string. Let *A* be the set of subscripts that follow s_n. That is, for all *s* in *A*:

a) CO(s_n,*s*) = *s* and
b) $D(NAME($s_1$, s_2, ..., s_{n-1}, *s*)) is not zero.

Then \$ORDER(NAME(s_1, s_2, ..., s_n)) returns that value *t* in *A* such that CO(*t*,*s*) = *s* for all *s* not equal to *t*; that is, all other subscripts which follow s_n 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, ..., s_n) where s_n may be the empty string. Let A be the set of subscripts that precede s_n. That is, for all *s* in A:

1) CP(s_n, *s*) = *s* and
2) \$D(NAME(s_1, s_2, ..., s_{n-1}, *s*)) is not zero.

Then \$ORDER(NAME(s_1, s_2, ..., s_n), -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 which precede *s* also precede *t*.

If no such *t* exists, \$ORDER(NAME(s_1, s_2, ..., s_n), -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] ($\underline{\text{expr}}_1$, $\underline{\text{expr}}_2$)

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 mth occurrence of the string d in s.

Let s be the value of $\underline{\text{expr}}_1$, and let d be the value of $\underline{\text{expr}}_2$. \$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] ($\underline{\text{expr}}_1$, $\underline{\text{expr}}_2$, <u>intexpr</u>)

Let m be the integer value of <u>intexpr</u>. \$PIECE(s,d,m) returns the substring of s bounded by but not including the m-1th and the mth occurrence of d.

\$P(s,d,m) = \$E(s,NF(s,d,m-1),NF(s,d,m) - \$L(d) - 1).

\$P[IECE] ($\underline{\text{expr}}_1$, $\underline{\text{expr}}_2$, $\underline{\text{intexpr}}_1$, $\underline{\text{intexpr}}_2$)

Let n be the integer value of $\underline{\text{intexpr}}_2$. \$PIECE(s,d,m,n) returns the substring of s bounded on the left but not including the m-1th occurrence of d in s, and bounded on the right but not including the nth 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] (<u>namevalue</u>)

See 7.1.4.12 for the definition of <u>namevalue</u>.

This form returns a value which is derived from <u>namevalue</u>. If <u>namevalue</u> has the form NAME(s_1, s_2, ..., s_n), considering n to be zero if there are no subscripts, then the function returns n.

Note that the <u>namevalue</u> is not "executed", and will not affect the naked indicator, nor generate an error if the <u>namevalue</u> represents an undefined <u>glvn</u>. The naked indicator will only be affected by the last <u>gvn</u> 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 , ... , s_n), 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 International 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 s_m.

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, ..., s_q) where s_q 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(s_1, s_2, ..., s_q, "").

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.

If the value of $QUERY(glvn) 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. [See note in Foreward].

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 command, the commandword 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 command 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 command executed.

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

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

The location is in the form:

```
place SP + eoffset

place    ::= | [ label ] [ + intlit ] [ ^ | environment | 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 command or line 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     ::=   | entryref                   |
                          | @ 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 entryref, the line denoted by the entryref is specified. If entryref 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

$TR[ANSLATE] (expr$_1$, expr$_2$)

Let *s* be the value of expr$_1$, $TRANSLATE(expr$_1$,expr$_2$) returns an edited form of *s* in which all characters in *s* which are found in expr$_2$ are removed.

$TR[ANSLATE] (expr$_1$, expr$_2$, expr$_3$)

Let *s* be the value of expr$_1$, $TRANSLATE(expr$_1$,expr$_2$,expr$_3$) returns an edited form of *s* in which all characters in *s* which are found in expr$_2$ are replaced by the positionally corresponding character in expr$_3$. If a character in *s* appears more than once in expr$_2$ 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 expr$_2$ remain unchanged. Characters in expr$_2$ which have no corresponding character in expr$_3$ are deleted from *s* (this is the case when expr$_3$ is shorter than expr$_2$).

Note: If the value of expr$_2$ 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.

7.2 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, 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 a svn with an undefined value causes an error condition with ecode="M8".

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 <u>ecode</u>="M9".
\	produces the integer interpretation of the result of the algebraic quotient.
#	produces the value of the left operand modulo the right argument. 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*.

**	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 <u>truthop</u>

```
truthop    ::=  | relation  |
                | logicalop |
```

7.2.2.1 Relational operator <u>relation</u>

```
                | =  |
                | <  |
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 '<u>relation</u> are defined by:

A '<u>relation</u> *B* has the same value as '(*A* <u>relation</u> *B*).

7.2.2.2 Numeric relations

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

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 *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 <u>lvn</u>..

7.2.2.4 Logical operator <u>logicalop</u>

```
    logicalop    ::=   | & |
                       | ! |
```

The operators ! and & are called logical operators. (They are given the names *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* = (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*)

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

$S = S_1 \ S_2 \ldots S_n$

such that there is a one-to-one order-preserving correspondence between the S_i and the pattern atoms, and each S_i *satisfies* its respective pattern atom. Note that some of the S_i 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 S_i 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     | strlit       |
                                | alternation  |

                   | intlit                          |
repcount     ::=   |                                 |
                   | [ intlit₁ ]  .  [ intlit₂ ]     |

                   |  Y patnonY Y   |
patcode      ::=   |  Z patnonZ Z   |   ...
                   |   patnonYZ     |

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

alternation  ::=   ( patatom [ , patatom ] ... )
```

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

Patcodes differing only in the use of corresponding upper and lower case letters are equivalent. Each patcode is satisfied by any single character in the union of the classes of characters represented, each class denoted by its own patcode letter. 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 patatom components individually matches the corresponding S_i.

Each strlit is satisfied by, and only by, the value of strlit.

If repcount has the form of an indefinite multiplier ".", patatom is satisfied by a concatenation of any number of S_i (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 S_i, each of which meets the specification of patatom. In particular, if the value of intlit is zero, the corresponding S_i is empty.

If repcount has the form of a range, intlit$_1$.intlit$_2$, 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 .intlit$_2$, the lower bound is taken to be zero. If the upper bound is omitted, so that the range has the form intlit$_1$. , the upper bound is taken to be indefinite; that is, the range is at least intlit$_1$ occurrences. Then patatom is satisfied by the concatenation of a number of S_i, 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:

$A \text{ '? } B \ = \ \text{'}(A \ ? \ B)$

8 Commands

8.1 General command rules

Every command starts with a commandword which dictates the syntax and interpretation of that command instance. commandwords differing only in the use of corresponding upper and lower case letters are equivalent. The International Standard contains the following commandwords:

```
                      | B[REAK]
                      | C[LOSE]
                      | D[O]
                      | E[LSE]
                      | F[OR]
                      | G[OTO]
                      | H[ALT]
                      | H[ANG]
                      | I[F]
                      | J[OB]
                      | K[ILL]
                      | L[OCK]
                      | M[ERGE]
commandword   ::=     | 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 commandwords other than those starting with the letter "Z" are reserved for future enhancement of the International Standard.

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

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

```
                  | syntax of BREAK command               |
                  | syntax of CLOSE command               |
                  |           .                           |
command     ::=   |           .                           |
                  |           .                           |
                  | syntax of XECUTE command              |
                  | syntax of Z[unspecified] command      |
```

For all commands allowing multiple arguments, the form

commandword arg_1, arg_2, ... arg_n

is equivalent in execution to

commandword arg_1 commandword arg_2 ... commandword arg_n

Within a command, 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).

An error condition occurs, with ecode="M11", when execution begins of any formalline unless that formalline has just been reached as a result of an exvar, an exfunc, a JOB command jobargument, or a DO command doargument that contains an actuallist.

8.1.1 Spaces in commands

Spaces are significant characters. The following rules apply to their use in lines.

a) There may be a space immediately preceding eol only if the line ends with a comment. (Since ls may immediately precede eol, this rule does not apply to the space which may stand for ls.)

b) If a command instance contains at least one argument, the commandword or postcond is followed by exactly one space; if the command is not the last of the line, or if a comment follows, the command is followed by one or more spaces.

c) If a command instance contains no argument and it is not the last command of the line, or if a comment follows, the commandword or postcond is followed by at least two spaces; if it is the last command of the line and no comment follows, the commandword or postcond is immediately followed by eol.

8.1.2 Comment comment

If a semicolon appears in the commandword initial-letter position, it is the start of a comment. The remainder of the line to eol must consist of graphics only, but is otherwise ignored and nonfunctional.

8.1.3 Command argument indirection

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

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

commandword SP L argument

the argument syntax will be expressed as

```
argument    ::=  | individual argument syntax |
                 |                            |
                 | @ 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 command 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 commands except ELSE, FOR, and IF may be made conditional as a whole by following the commandword immediately by the post-conditional postcond.

```
postcond    ::=  [ : tvexpr ]
```

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

The postcond may also be used to conditionalize the arguments of DO, GOTO, 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 Command timeout timeout

The OPEN, LOCK, JOB, and READ commands employ an optional timeout specification, associated with the testing of an external condition.

```
timeout    ::=   : numexpr
```

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

$TEST will not be altered if the timeout is absent.

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 true, $TEST is set to 1; otherwise, $TEST is set to 0. 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 true, $TEST is set to 1; otherwise, $TEST is set to 0.

8.1.6 Line reference lineref

The DO , GOTO, and JOB commands, extrinsic functions and extrinsic variables, as well as the $TEXT function, contain in their arguments means for referring to particular lines within any routine. This subclause describes the means for making line references.

A reference to a line is either an entryref or a labelref. An entryref allows the specification of integer offsets from a label (eg, LOOP+5 references the fifth line after the line that has LOOP for a label). Also, an entryref allows indirection of both the label and the routinename. A labelref, on the other hand, allows neither label offsets nor indirection.

```
lineref    ::=  | entryref |
                | labelref |
```

8.1.6.1 Entry reference entryref

The total line specification in DO, GOTO, JOB, and $TEXT is in the form of entryref.

```
                  | dlabel [ + intexpr ] [ ^ routineref ] |
entryref    ::=   |                                        |
                  |            ^ routineref                |
```

If the routine reference (^ routineref) is absent, the routine being executed is implied. 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". When label is an instance of intlit, leading zeros are significant to its spelling.

In the context of DO , GOTO, or JOB, either of the following conditions causes an error condition with ecode="M13".

a) A value of intexpr so large as not to denote a line within the bounds of the given routine.

b) A spelling of label which does not occur in a defining occurrence in the given routine.

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.

DO, GOTO, and JOB commands, as well as 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.

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.

```
                  |        label         |
dlabel      ::=   |                      |
                  | @ expratom V dlabel  |

                  | [ | environment | ] routinename |
routineref  ::=   |                                 |
                  | @ expratom V routineref         |
```

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".

8.1.6.2 Label reference labelref

When the DO or JOB commands or exfunc or exvar include parameters to be passed to the specified routine, the +intexpr form of entryref is not permitted and the specified line must be a formalline. The line specification labelref is used instead:

```
                  | label [ ^ [ | environment | ] routinename ] |
labelref    ::=   |                                              |
                  | ^ [ | environment | ] routinename            |
```

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 command, 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".

8.1.6.3 External reference externref

```
        externref    ::=   & [ packagename . ] externalroutinename

       packagename   ::=   name

externalroutinename  ::= name [ ^ name ]
```

The ampersand (&) character designates a program whose namespace is external to the current M 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.

8.1.7 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 command with an actuallist, or to a process as the result of a JOB command with an actuallist.

```
        actuallist  ::=   ( [ L actual ] )

                         ┌                  ┐
                         │ . actualname     │
        actual      ::=  │                  │
                         │ expr             │
                         └                  ┘

                         │ name                        │
        actualname  ::=  │                             │
                         │ @ expratom V actualname     │
```

When parameter passing occurs, the formalline 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.

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 actuallist name. If an entry is found, the parameter list item is the DATA-CELL pointer in this NAME-TABLE entry. If the actuallist 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 null, 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 commands may.

Execution is then initiated at the first command following the ls of the line specified by the labelref. Execution of the subroutine continues until an eor or a QUIT is executed that is not within the scope of a subsequently executed doargument, argumentless DO, xargument, exfunc, exvar, or FOR. In the case of an exfunc or exvar, the subroutine must be terminated by a QUIT with an argument.

At the time of the QUIT, the formallist names are unbound and the original variable environment is restored. See 8.2.16 for a discussion of the semantics of the QUIT operation.

When calling to an externref, pass-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 command. The exact mechanism performing this operation is unspecified.

b) The resultant events are unspecified, if the data in the M 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.

8.2 Command definitions

The specifications of all commands follow.

8.2.1 BREAK

```
B[REAK] postcond | [ SP ]                       |
                 | argument syntax unspecified  |
```

BREAK provides an access point within the International 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 SP L closeargument

                    | expr [ : deviceparameters ]  |
    closeargument ::= |                              |
                    | @ expratom V L closeargument  |

                      | 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.

The value of the first expr of each closeargument identifies a device (or *file* or *data set*). The interpretation of the value of this expr is left to the implementor. 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 command 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, the implementor may choose to execute implicitly the commands OPEN *P* USE *P*, where *P* designates a predetermined default device. If the implementor chooses otherwise, $IO is given the value of the empty string.

8.2.3 DO

```
                    | [ SP ]              |
D[O] postcond       |                     |
                    | SP L doargument     |

                       | entryref postcond                   |
                       |                                     |
doargument ::=         | labelref actuallist postcond        |
                       |                                     |
                       | externref [ actuallist ] postcond   |
                       |                                     |
                       | @ expratom V L doargument           |
```

An argumentless DO initiates execution of an inner block of lines. If postcond is present and its tvexpr is false, the execution of the command is complete. If postcond is absent, or the postcond is present and its tvexpr is true, the DO places a DO frame containing the current execution location, the current execution level, and the current value of $TEST on the PROCESS-STACK, increases the execution level by one, and continues execution at the next line in the routine. (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, argumentless DO, 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 command (if any) following the argumentless DO.

DO with arguments is a generalized call to the subroutine specified by the entryref, the labelref, or the externref in each doargument. The line specified by the entryref or labelref, must have a LEVEL of one. If the line specified is an externref then an implicit LEVEL of 1 is assumed, unless otherwise specified within the binding. Execution of a doargument to a line whose LEVEL is not one causes an error condition with ecode="M14".

If the actuallist is present in an executed doargument, parameter passing occurs and the formalline 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 and the execution level are 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 command position specified by the reference as follows:

1) For entryref and labelref, this is the first command that follows the ls of the line specified by entryref or labelref. Execution of the subroutine (within the M environment) continues until an eor or a QUIT is executed that is not within the scope of a subsequently executed FOR, argumentless DO, doargument, xargument, exfunc, or exvar. The scope of this internally referenced doargument is said to extend to the execution of that QUIT or eor. (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.

8.2.4 ELSE

```
E[LSE] [ SP ]
```

If the value of $TEST is 1, the remainder of the line to the right of the ELSE is not executed. If the value of $TEST is 0, execution continues normally at the next command.

8.2.5 FOR

```
F[OR] | [ SP ]                  |
      |                         |
      | SP lvn = L forparameter |

                    | expr                               |
forparameter ::=    | numexpr1 : numexpr2 : numexpr3     |
                    | numexpr1 : numexpr2                |
```

The *scope* of the FOR command begins at the next command following the FOR on the same line and ends just prior to the eol on this line.

The FOR with arguments specifies repeated execution of the commands 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 commands in the scope is specified as follows. (Note that *A*, *B*, and *C* are hidden temporaries.)

a) If the forparameter is of the form expr$_1$.

1) Set lvn = expr.
2) Execute the commands in the scope once.
3) Processing of this forparameter is complete.

b) If the forparameter is of the form numexpr$_1$: numexpr$_2$: numexpr$_3$ and numexpr$_2$ is nonnegative.

1) Set A = numexpr$_1$.
2) Set B = numexpr$_2$.
3) Set C = numexpr$_3$.
4) Set lvn = A.
5) If lvn > C, processing of this forparameter is complete.
6) Execute the commands 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 numexpr$_1$: numexpr$_2$: numexpr$_3$ and numexpr$_2$ is negative.

1) Set A = numexpr$_1$.
2) Set B = numexpr$_2$.
3) Set C = numexpr$_3$.
4) Set lvn = A.
5) If lvn < C, processing of this forparameter is complete.
6) Execute the commands 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 numexpr$_1$: numexpr$_2$.

1) Set A = numexpr$_1$.
2) Set B = numexpr$_2$.
3) Set lvn = A.
4) Execute the commands 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 command has no argument:

a) Execute the commands in the scope once; since no lvn has been specified, it cannot be referenced.

b) Goto a.

Note that form d. and the argumentless FOR, specify endless loops. Termination of these loops must occur by execution of a QUIT or GOTO within the scope of the FOR. These two termination methods are available within the scope of a FOR independent of the form of forparameter currently in control of the execution of the scope; they are described below. Note also that no forparameter to the right of one of form d. can be executed.

Note that if the scope of a FOR (the *outer* FOR) contains an *inner* FOR, one execution of the scope of commands of the outer FOR encompasses all executions of the scope of commands of the inner FOR corresponding to one complete pass through the inner FOR command's forparameter list.

Execution of a QUIT within the scope of a FOR has two effects.

a) It terminates that particular execution of the scope at the QUIT; commands to the right of 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 commands in the scope not to be executed under their control.

In other words, execution of QUIT effects the immediate termination of the innermost FOR whose scope contains the QUIT.

Execution of GOTO effects the immediate termination of all FOR commands in the line containing the GOTO, and it transfers execution control to the point specified. Note that the execution of a QUIT within the scope of a FOR does not affect the variable environment, e.g., stacked NEW frames are not removed or processed.

8.2.6 GOTO

```
G[OTO] postcond SP L gotoargument

                    | entryref postcond              |
gotoargument ::=    |                                |
                    | @ expratom V L gotoargument    |
```

GOTO is a generalized transfer of control. If provision for a return of control is desired, DO may be used.

Each gotoargument is examined, one at a time in left-to-right order, until the first one is found whose postcond is either absent, or whose postcond is present and its tvexpr is true. If no such gotoargument is found, control is not transferred and execution continues normally. If such a gotoargument is found, execution continues at the left of the line it specifies, provided the line has the same LEVEL as the line containing the GOTO and, if the LEVEL of the line containing the GOTO is greater than one, there may be no lines of lower execution LEVEL between the line specified by the gotoargument and the line containing the GOTO, and the line containing the GOTO and the line specified by the gotoargument must be in the same routine. Otherwise, an error occurs with ecode="M45".

8.2.7 HALT

```
H[ALT] postcond [ SP ]
```

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.

8.2.8 HANG

```
H[ANG] postcond SP 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.

8.2.9 IF

```
I[F]  | [ SP ]              |
      |                     |
      | SP L ifargument     |

                   | tvexpr                     |
ifargument ::=     |                            |
                   | @ expratom V L ifargument  |
```

In its argumentless form, IF is the inverse of ELSE. That is, if the value of $TEST is 0, the remainder of the line to the right of the IF is not executed. If the value of $TEST is 1, execution continues normally at the next command.

If exactly one argument is present, the value of tvexpr is placed into $TEST; then the function described above is performed.

IF with *n* arguments is equivalent in execution to *n* IF commands, each with one argument, with the respective arguments in the same order. This may be thought of as an implied *and* of the conditions expressed by the arguments.

8.2.10 JOB

```
J[OB] postcond SP L jobargument

                     | entryref [ : jobparameters ]                |
jobargument ::=      |                                             |
                     | labelref actuallist [ : jobparameters ]     |
                     |                                             |
                     | @ expratom V L jobargument                  |

                     | processparameters [ timeout ] |
jobparameters ::=    |                               |
                     | timeout                       |

                     | expr                             |
processparameters ::=|                                  |
                     | ( [ [ expr ] : ] ... expr )      |
```

For each jobargument, the JOB command attempts to initiate another M process. If the actuallist is present in a jobargument, the formalline 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 command initiates this process at the line 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 a timeout is present, the condition reported by $TEST is the success of initiating the process. If no timeout is present, the value of $TEST is not changed, and process execution is suspended until the process named in the jobargument is successfully initiated. The meaning of success in either context is defined by the implementation.

8.2.11 KILL

```
                          | [ SP ]                  |
      K[ILL] postcond     |                         |
                          | SP L killargument       |

                          | glvn                               |
   killargument  ::=      | ( L lname )                        |
                          | @ expratom V L killargument        |

                          | name                    |
      lname      ::=      |                         |
                          | @ expratom V name       |
```

The three argument 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*) where *V* is a glvn.

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 *V* is unsubscripted, delete all tuples in the DATA-CELL.

c) If *V* has subscripts, then let *N* be the number of subscripts in *V*. Delete all tuples in the DATA-CELL which have *N* or greater subscripts and whose first *N* subscripts are the same as those in *V*.

Note that as a result of procedure *K*, $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 *K* to the specified glvn.
b) Exclusive Kill	- apply *K* to all names in the NAME-TABLE except those in the argument list. Note that the names in the argument list of an exclusive kill may not be subscripted.
c) Kill All	- apply *K* to all names in the NAME-TABLE.

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 LOCK

```
                        | [ SP ]              |
L[OCK] postcond         |                     |
                        | SP L lockargument   |

                        |  ┌─┐  |   nref       |                |
                        |  |+|  |              |  [ timeout ]   |
lockargument  ::=       |  |-|  | ( L nref )   |                |
                        |  └─┘                                   |
                        | @ expratom V L lockargument            |

                        | [ ^ ] [ | environment | ] name [ ( L expr ) ] |
nref          ::=       |                                               |
                        | @ expratom V nref                             |
```

LOCK provides a generalized interlock facility available to concurrently executing M 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 M 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 $N(s_1,\ldots,s_i)$ where N has the same spelling as nref.

b) If the occurrence of nref is subscripted, let the nref be $N(s_1,s_2,\ldots,s_n)$. Then the subspace is the set of the following points: one point for $N(s_1,s_2,\ldots,s_n)$ and one point for each descendant (see 7.1.5.3 $DATA function for a definition of descendant) of nref.

If the LOCK command 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 commands outside the scope of a TRANSACTION may not OVERLAP.

b) All global variable modifications produced by the execution of commands 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 the value of $TEST is set to zero and 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 and $TEST is set to one.

If no timeout is present, then the value of $TEST is not affected by execution of the lockargument.

8.2.13 MERGE

```
M[ERGE] postcond SP L mergeargument
```

mergeargument ::= | glvn$_1$ = glvn$_2$ |
| @ expratom V L mergeargument |

MERGE provides a facility to copy a $glvn_2$ into a $glvn_1$ and all descendants of $glvn_2$ into descendants of $glvn_1$ according to the scheme described below.

MERGE does not KILL any nodes in $glvn_1$, or any of its descendants.

Assume that $glvn_1$ is represented as $A(i_1, i_2, ..., i_x)$ (x'<0) and that $glvn_2$ is represented as $B(j_1, j_2, ..., j_y)$ (y'<0).

Then:

a) If $DATA($B(j_1,j_2,...,j_y)$) has a value of 1 or 11, then the value of $glvn_2$ is given to $glvn_1$.

b) The value for every occurrence of z, such that z > 0 and $DATA($B(j_1, j_2,...,j_{y+z})$) has a value of 1 or 11, the value of $B(j_1,j_2,...,j_{y+z})$ is given to $A(i_1, i_2,...,i_x,j_{y+1},j_{y+2},...,j_{y+z})$.

The state of the naked indicator will be modified as if $DATA($glvn_2$)#10=1 and the command SET $glvn_1$=$glvn_2$ would have been executed.

If $glvn_1$ is a descendant of $glvn_2$ or if $glvn_2$ is a descendant of $glvn_1$ an error condition occurs with ecode="M19".

8.2.14 NEW

```
                         | [ SP ]                |
N[EW] postcond           |                       |
                         | SP 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 command 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 command argument, to point to empty DATA-CELLs.

c) Selective NEW copies the entry corresponding to the name specified by the command 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 command 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).

8.2.15 OPEN

```
        O[PEN] postcond SP L openargument

                         | expr [ : 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)
                               └       ┘
```

mnemonicspace specifies the set of controlmnemonics that may be used within format arguments to subsequent READ and WRITE commands. 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.

When a mnemonicspec contains a list of mnemonicspaces, the first one determines the active mnemonicspace, which may be changed by a USE command. If the device does not support the mnemonicspace, 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.

The value of the first expr of each openargument identifies a device (or *file* or *data set*). The interpretation of the value of this expr or of any exprs in deviceparameters is left to the implementor. (see 8.2.2 for the syntax specification of deviceparameters.)

The OPEN command 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 8.2.23)

For each openargument, the OPEN command 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 a timeout is present, the condition reported by $TEST is the success of obtaining ownership. If no timeout is present, the value of $TEST is not changed and process execution is suspended until seizure of ownership has been successfully accomplished by the process that issued the OPEN command.

Ownership is relinquished by execution of the CLOSE command. 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.

8.2.16 QUIT

```
                 | [ SP ]                 |
Q[UIT] postcond  | SP expr                |
                 | SP @ expratom V expr   |
```

QUIT terminates execution of an argumentless DO command, doargument, xargument, exfunc, exvar, or FOR command.

Encountering the end-of-routine mark eor 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 argumentless DO command, 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 an argumentless DO command, doargument, xargument, exfunc, or exvar is to restore the previous variable environment (if necessary), restore the value of $TEST (if necessary), restore the previous execution level, and continue execution at the location of the invoking argumentless DO command, 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.

Finally, copy all NAME-TABLE entries from this frame into the NAME-TABLE.

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) If the frame is from an exfunc or exvar or from an argumentless DO command, set the value of $TEST to the value saved in the frame.

g) Restore the execution level and continue execution at the location specified in the frame.

8.2.17 READ

```
R[EAD] postcond SP L readargument

                      | strlit                                 |
                      | format                                 |
readargument ::=      | glvn [ readcount ] [ timeout ]         |
                      | * glvn [ timeout ]                     |
                      | @ expratom V L readargument            |
```

See 8.2.25 for a definition of format.

```
        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 8.2.23 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.

When a timeout is present, $TEST is affected as follows. If the input message has been terminated at or before the time at which execution resumes, $TEST is set to 1; otherwise, $TEST is set to 0.

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 command 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)

8.2.18 SET

```
    S[ET] postcond SP L setargument

    setargument ::=     | setdestination = expr       |
                        | @ expratom V L setargument  |

 setdestination ::=     | setleft       |
                        | ( L setleft ) |

                        | leftrestricted |
      setleft   ::=     | leftexpr       |
                        | glvn           |

                        | $D[EVICE] |
                        | $K[EY]    |
 leftrestricted ::=     | $X        |
                        | $Y        |

                        | setpiece   |
      leftexpr  ::=     | setextract |
                        | setev      |

      setpiece  ::=  $P[IECE] ( glvn , expr₁ [ , intexpr₁ [ , intexpr₂ ] ] )

         setextract ::=  $E[XTRACT] ( glvn [ , intexpr₁ [ , intexpr₂ ] ] )

         setev      ::=  | $EC[ODE] |
                         | $ET[RAP] |
```

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, the glvn that is the first argument of the setpiece 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 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 command, the various setlefts may receive different values.

c) One of the following four 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 *m*th through the *n*th 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.

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 command. 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.

8.2.19 TCOMMIT

```
TC[OMMIT] postcond [ SP ]
```

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.

8.2.20 TRESTART

```
TRE[START] postcond [ SP ]
```

If $TLEVEL is greater than zero, TRESTART performs a RESTART.

If $TLEVEL is zero, TRESTART generates an error with ecode="M44".

8.2.21 TROLLBACK

```
TRO[LLBACK] postcond [ SP ]
```

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".

8.2.22 TSTART

```
                          | [SP]                                |
TS[TART] postcond         | SP tstartargument                   |
                          | SP @ expratom V tstartargument      |

  tstartargument ::= [ restartargument ] [ : transparameters ]

                          | lname          |
restartargument ::=       | ( L lname )    |
                          | *              |
                          | ( )            |

transparameters ::=       | tsparam                        |
                          | ( tsparam [ : tsparam ] ... )  |

       tsparam      ::=  tstartkeyword [ = expr ]
```

tstartkeywords that differ only in the use of corresponding upper and lower-case letters are equivalent. The International Standard defines the following keywords:

```
S[ERIAL]
T[RANSACTIONID] = expr
Z[unspecified] [ = expr ]
```

Unused keywords other than those starting with the letter "Z" are reserved for future enhancement of the International 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 command, values for $TEST and 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 command 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.

8.2.23 USE

```
U[SE] postcond SP L useargument

                    |       [ : deviceparameters                                ] |
                    | expr  [                                                   ] |
useargument ::=     |       [ : [ deviceparameters ] : mnemonicspace            ] |
                    |                                                             |
                    | @ expratom V L useargument                                  |
```

See 8.2.15 OPEN for mnemonicspace.

The value of the first expr of each useargument identifies a device (or *file* or *data set*). The interpretation of the value of this expr or of any exprs in deviceparameters is left to the implementor. (see 8.2.2 for the syntax specification of 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 command, 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 command.

The specified device remains current until such time as a new USE command is executed. As a side effect of employing expr to designate a current device, $IO is given the value of expr.

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 command, 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 command, 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.

8.2.24 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 command may not be portable.

8.2.25 WRITE

```
        W[RITE] postcond SP 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.

```
                            | | ! |                                  |
        format       ::=    | | # | ... [ ? intexpr ]                |
                            |                                        |
                            | ? 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 command. 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.

8.2.26 XECUTE

```
X[ECUTE] postcond SP L xargument

                    | expr postcond               |
xargument   ::=     |                             |
                    | @ expratom V L xargument    |
```

XECUTE provides a means of executing M code which 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 *y*, where *y* is the spelling of an otherwise unused label attached to the following two-line subroutine considered to be a part of the currently executing routine:

```
y  ls   x    eol

ls  QUIT  eol
```

8.2.27 Z

```
Z[unspecified] arguments unspecified
```

All commandwords in a given implementation which are not defined in the International Standard are to begin with the letter Z. This convention protects the International Standard for future enhancement.

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     ::=  name
```

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 M 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 International Standard.

Portability Requirements

10 Character set

The character set used for routines and data is restricted to the Character Set Profile M (as defined in Annex A).

11 Expression elements

11.1 Names

The use of ident in names is restricted to upper case alphabetic characters. While there is no explicit limit on name length, only the first eight characters are uniquely distinguished. This length restriction places an implicit limit on the number of unique names.

11.2 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.

11.3 Local variables

11.3.1 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).

11.3.2 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 the maximum string length (see 2.8). The length of an array reference is calculated as follows:
assuming an array reference in the form
name(i_1,i_2, ... ,i_n)

N = \$L(name)

I = \$L(i_1) + \$L(i_2) + ... + \$L(i_n)

where each subscript (i_1 through i_n) is either a numlit or a sublit

L = n

then the total length of an array reference is

N + I + (2 * L) + 15

11.3.3 Values of subscripts

Local variable subscript values are nonempty strings which shall only contain characters from the M printable character subset. There is no specific restriction on the length of a subscript, but 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.

11.4 Global variables

11.4.1 Number of global variables

There is no explicit limit on the number of distinct global variable names in existence at any time.

11.4.2 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 the maximum string length (see 2.8). The length of an array reference is calculated as follows:
assuming an array reference in the form:

^|<u>environment</u>|<u>name</u>(i_1,i_2, ... ,i_n)

```
        E = $L(environment)
        N = $L(name)
        I = $L(i1) + $L(i2) + ... + $L(in)
```

where each subscript (i_1 through i_n) is either a <u>numlit</u> or a <u>sublit</u>

```
        L = n
```

then the total length of an array reference is

```
        E + 3 + N + I + ( 2 * L ) + 15
```

11.4.3 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).

11.4.4 Number of nodes

There is no explicit limit on the number of distinct global variable nodes which may be defined.

11.5 Data types

The M 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 M language specification. For example, integers might be stored as binary integers and converted to decimal character strings whenever an operation requires a string value.

11.6 Number range

All values used in arithmetic operations or in any context requiring a numeric interpretation are within the inclusive intervals [-10^{25}, -10^{-25}] or [10^{-25}, 10^{25}], 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 10^{15}. The error introduced by exponentiation shall not exceed one part in 10^{7}.

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.

11.7 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).

11.8 Character strings

Character string length is limited to 255 characters. The characters permitted within character strings must include those defined in the ASCII standard (ANSI X3.4-1986).

11.9 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 8.2.25 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).

12 Expressions

12.1 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).

12.2 Results

Any result, whether intermediate or final, which does not satisfy the constraints on character strings (see 2.8) is erroneous. Furthermore, integer results are erroneous if they do not satisfy the constraints on integers (see 2.7).

12.3 External references

External references are not portable.

13 Routines and command lines

13.1 Command lines

A command line (line) must satisfy the constraints on character strings (see 2.8). The length of a command line is the number of characters in the line up to but not including the eol.

The characters within a command 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).

13.2 Number of command lines

There is no explicit limit on the number of command lines in a routine, subject to storage space restrictions (Clause 8).

13.3 Number of commands

The number of commands per line is limited only by the restriction on the maximum command line length (see 4.1).

13.4 Labels

A label of the form name is subject to the constraints on names; labels of the form intlit are subject to the length constraint on names (see 2.1).

13.5 Number of labels

There is no explicit limit on the number of labels in a routine. However, the following restrictions apply:

a) A command line may have only one label.

b) No two lines may be labeled with equivalent (not uniquely distinguishable) labels.

13.6 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).

14 External routine calls

When the external routine called is not within the current default M 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.

15 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.

16 Indirection

The values of the argument of indirection and the argument of the XECUTE command are subject to the constraints on character string length (see 2.8). They are additionally restricted to the character set limitations of command lines (see 4.1).

17 Storage space restrictions

The size of a single routine must not exceed 10,000 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 10,000 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.

18 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).

19 Formats

Device control may be effected through the READ and WRITE commands 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.

19.1 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.

19.2 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.

19.3 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.

20 Transaction processing

20.1 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.

20.2 Number of nested TSTARTs within a TRANSACTION

A single TRANSACTION must not contain more than 126 TSTARTs after the TSTART that initiates the TRANSACTION.

21 Other portability requirements

Programmers should exercise caution in the use of noninteger values for the HANG command and in timeouts. In general, the period of actual time which elapses upon the execution of a HANG command 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 7.1.3.9 when processid is not their own $JOB.

ANSI X3.64

22 The binding

ANSI X3.64 is accessed from the M 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 command. If a READ command is used, the argument list in the statement(s) must be ordered to correctly accept the returned values. If a WRITE command is used the values returned may be read by a single, or series of, READ commands. These READ commands 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 M 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.

22.1 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.

22.2 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 command that is executed after the control-function has been issued.

22.3 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

22.4 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.

23 Portability issues

23.1 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.

23.2 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 Command
DA	Device Attributes
DCS	Device Control String
FNT	Font Selection
INT	Interrupt
OSC	Operating System Command
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

24 Conformance

Each implementation must supply a list of the controlmnemonic and arguments that are supported for each device.