

Information technology — Programming languages —M

ICS 35.080

National foreword

This British Standard reproduces verbatim ISO/IEC 11756:1999 and implements it as the UK national standard. It supersedes BS ISO/IEC 11756:1992 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming Languages, Their Environments and System Software Interfaces, which has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

A list of organizations represented on this committee can be obtained on request to its secretary.

Cross-references

The British Standards which implement international or European publications referred to in this document may be found in the BSI Standards Catalogue under the section entitled “International Standards Correspondence Index”, or by using the “Find” facility of the BSI Standards Electronic Catalogue.

A British Standard does not purport to include all the necessary provisions of a contract. Users of British Standards are responsible for their correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

Summary of pages

This document comprises a front cover, an inside front cover, the ISO/IEC title page, pages ii to vii, a blank page, pages 1 to 115 and a back cover.

The BSI copyright notice displayed in this document indicates when the document was last issued.

This British Standard, having been prepared under the direction of the DISC Board, was published under the authority of the Standards Committee and comes into effect on 15 November 1999

© BSI 11 1999

ISBN 0 580 35494 6

Amendments issued since publication

Amd. No.	Date	Comments

INTERNATIONAL STANDARD

ISO/IEC 11756

Second edition
1999-06-01

Information technology — Programming languages — M

Technologies de l'information — Langages de programmation — M



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 <u>routine</u>	9
6.1 Routine head <u>routinehead</u>	9
6.2 Routine body <u>routinebody</u>	10
6.2.1 Level line <u>levelline</u>	10
6.2.2 Formal line <u>formalline</u>	10
6.2.3 Label <u>label</u>	10
6.2.4 Label separator <u>ls</u>	11
6.2.5 Line body <u>linebody</u>	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 <u>expr</u>	15
7.1 Expression atom <u>expratom</u>	15
7.1.1 Variables	15
7.1.2 Variable name <u>glvn</u>	16
7.1.2.1 Local variable name <u>lvn</u>	16
7.1.2.2 Local variable handling	16
7.1.2.3 Process-stack	19
7.1.2.4 Global variable name <u>gvn</u>	20
7.1.3 Structured system variable <u>ssvn</u>	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 <u>ssvns</u> specifying default environments	25
7.1.4 Expression item <u>expritem</u>	26
7.1.4.1 String literal <u>strlit</u>	26

7.1.4.2 Numeric literal <u>numlit</u>	27
7.1.4.3 Numeric data values	27
7.1.4.4 Meaning of <u>numlit</u>	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 <u>exfunc</u>	29
7.1.4.9 Extrinsic special variable <u>exvar</u>	30
7.1.4.10 Intrinsic special variable names <u>svn</u>	30
7.1.4.11 Unary operator <u>unaryop</u>	34
7.1.4.12 Name value <u>namevalue</u>	35
7.1.5 Intrinsic function <u>function</u>	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 <u>exptail</u>	48
7.2.1 Binary operator <u>binaryop</u>	48
7.2.1.1 Concatenation operator	49
7.2.1.2 Arithmetic binary operators	49
7.2.2 Truth operator <u>truthop</u>	49
7.2.2.1 Relational operator <u>relation</u>	49
7.2.2.2 Numeric relations	50
7.2.2.3 String relations	50
7.2.2.4 Logical operator <u>logicalop</u>	50
7.2.3 Pattern match <u>pattern</u>	51
8. Commands	52
8.1 General command rules	52
8.1.1 Spaces in commands	53
8.1.2 Comment <u>comment</u>	54
8.1.3 Command argument indirection	54
8.1.4 Post conditional <u>postcond</u>	54
8.1.5 Command timeout <u>timeout</u>	55
8.1.6 Line reference <u>lineref</u>	55
8.1.6.1 Entry reference <u>entryref</u>	55
8.1.6.2 Label reference <u>labelref</u>	56
8.1.6.3 External reference <u>externref</u>	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 <u>charset</u>	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 <u>mnemonicspace</u>	86
19.2 <u>controlmnemonic</u>	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.

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 *n*th 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. Metalinguage description

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

<u>Operator</u>	<u>Meaning</u>
<code>::=</code>	definition
<code>[]</code>	option
<code> </code>	grouping
<code>...</code>	optional indefinite repetition
<u>L</u>	list
<u>V</u>	value
<u>SP</u>	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:

$$\text{name ::= } \left| \begin{array}{c} \% \\ \text{ident} \end{array} \right| \left[\begin{array}{c} \text{digit} \\ \text{ident} \end{array} \right] \dots$$

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

$$\text{exampleargument ::= } \left| \begin{array}{c} \text{expr} \\ @ \text{expratom V L exampleargument} \end{array} \right|$$

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 ::=

%
<u>ident</u>

[

<u>digit</u>
<u>ident</u>

] ...

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 ::=

<u>levelline</u>
<u>formalline</u>

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 ::=

<u>name</u>
<u>intlitt</u>

See 7.1.4.2 for a definition of intlitt.

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.

$$\text{linebody} ::= \left[\begin{array}{l} \text{commands [cs comment]} \\ \text{[commands cs] extsyntax} \\ \text{comment} \end{array} \right] \text{eol}$$

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 formalist 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) \$RESTART 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.

$$\underline{\text{expr}} \quad ::= \quad \underline{\text{expratom}} \quad [\quad \underline{\text{exprtail}} \quad] \quad \dots$$

7.1 Expression atom expratom

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

$$\underline{\text{expratom}} \quad ::= \quad \left| \begin{array}{l} \underline{\text{glvn}} \\ \underline{\text{expritem}} \end{array} \right|$$

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.

$$\underline{glvn} ::= \left| \begin{array}{l} \underline{lvn} \\ \underline{gvn} \\ \underline{ssvn} \end{array} \right|$$

7.1.2.1 Local variable name lvn

$$\underline{lvn} ::= \left| \begin{array}{l} \underline{rlvn} \\ @ \underline{expratom} \text{ V } \underline{lvn} \end{array} \right|$$

$$\underline{rlvn} ::= \left| \begin{array}{l} \underline{name} [(\text{L } \underline{expr})] \\ @ \underline{lnamind} @ (\text{L } \underline{expr}) \end{array} \right|$$

$$\underline{lnamind} ::= \underline{rexpratom} \text{ V } \underline{lvn}$$

$$\underline{rexpratom} ::= \left| \begin{array}{l} \underline{rlvn} \\ \underline{rgvn} \\ \underline{exprite} \end{array} \right|$$

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

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 l_{vn} is said to be *undefined*.

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

```
VAR1 = "Hello"
VAR2 = 12.34
VAR3 = "abc"
VAR3("Smith","John",1234)=123
VAR3("Widget","red") = -56
```

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

NAME-TABLE	VALUE-TABLE DATA-CELLS
VAR1----->	() = "Hello"
VAR2----->	() = 12.34
XYZ----->	
VAR3----->	() = "abc" ("Smith", "John", 1234) = 123 ("Widget", "red") = -56
DEF----->	

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

- If the name of the l_{vn} 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.
- Otherwise, the pointer in the NAME-TABLE entry which contained the name of the l_{vn} is extracted. The operations in steps c and d refer to tuples in that DATA-CELL referred to by this pointer.
- If the l_{vn} 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.
- If the l_{vn} is subscripted, then the tuple of subscripts in the DATA-CELL (i.e., the tuple created by dropping the name of the l_{vn}; 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

<u>gvn</u>	::=	$\left \begin{array}{l} \underline{rgvn} \\ @ \underline{expratom} \vee \underline{gvn} \end{array} \right $
<u>rgvn</u>	::=	$\left \begin{array}{l} ^{(\underline{L} \underline{expr})} \\ ^{ [\mid \underline{environment} \mid] \underline{name} [(\underline{L} \underline{expr})] } \\ @ \underline{gnamind} @ (\underline{L} \underline{expr}) \end{array} \right $
<u>gnamind</u>	::=	$\underline{rexpratom} \vee \underline{gvn}$
<u>environment</u>	::=	<u>expr</u>

The prefix $\wedge$ 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

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

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

$$\wedge(s_1, s_2, \dots, s_i) \quad (i \text{ positive})$$

results in a global reference of the form

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

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

<u>ssvn</u> ::=	syntax of <u>^\$CHARACTER</u> structured system variable syntax of <u>^\$DEVICE</u> structured system variable syntax of <u>^\$GLOBAL</u> structured system variable syntax of <u>^\$JOB</u> structured system variable syntax of <u>^\$LOCK</u> structured system variable syntax of <u>^\$ROUTINE</u> structured system variable syntax of <u>^\$SYSTEM</u> structured system variable syntax of <u>^\$Z[unspecified]</u> 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[CHARACTER] (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(charsetexpr₁ , expr V "INPUT" , charsetexpr₂) = expr V algoref

<u>algoref</u> ::=		<u>emptystring</u>	
		\$\$ <u>labelref</u>	
		\$& <u>externref</u>	
		\$ <u>functionname</u>	

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

Output-Transformation:

$\text{\textasciix}\$ \text{CHARACTER}(\text{charsetexpr}_1, \text{expr} \vee \text{"OUTPUT"}, \text{charsetexpr}_2) = \text{expr} \vee \text{algoref}$

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

Valid name characters:

$\text{\textasciix}\$ \text{CHARACTER}(\text{charsetexpr}, \text{expr} \vee \text{"IDENT"}) = \text{expr} \vee \text{algoref}$

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:

$\text{\textasciix}\$ \text{CHARACTER}(\text{charsetexpr}, \text{expr} \vee \text{"PATCODE"}, \text{expr} \vee \text{patcode}) = \text{expr} \vee \text{algoref}$

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:

$\text{\^{\$}CHARACTER}(\text{charsetexpr}, \text{expr} \vee \text{"COLLATE"}) = \text{expr} \vee \text{alгореf}$

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

7.1.3.2 $\text{\^{\$}DEVICE}$

$\text{\^{\$}D[EVICE]}(\text{deviceexpr})$

$\text{deviceexpr} ::= \text{expr} \vee \text{device}$

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

$\text{\^{\$}DEVICE}$ 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 $\text{\^{\$}DEVICE}$.

Device characteristic information for a device is stored beneath the $\text{\^{\$}DEVICE}(\text{deviceexpr})$ node:

$\text{\^{\$}DEVICE}(\text{deviceexpr}, \text{expr} \vee \text{"CHARACTER"}) = \text{charsetexpr}$

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.

$\text{\^{\$}DEVICE}(\text{deviceexpr}, \text{expr} \vee \text{deviceattribute})$

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 $\text{\^{\$}DEVICE}$. 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 $\text{\^{\$}GLOBAL}$

$\text{\^{\$}G[LOBAL]}(\text{gvnexpr})$

$\text{gvnexpr} ::= \text{expr} \vee \text{gvn}$

$\text{\^{\$}GLOBAL}$ provides information about the existence and characteristics of globals.

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

$\text{\^{\$}GLOBAL}(\text{gvnexpr}, \text{expr} \vee \text{"CHARACTER"}) = \text{charsetexpr}$

This node identifies the Character Set Profile of the specified global. When the first node in a global is created, and the node $\text{\^{\$}GLOBAL}(\text{gvnexpr}, \text{"CHARACTER"})$ has a \$DATA value of zero, the value assigned is that of $\text{\^{\$}JOB}(\text{\$JOB}, \text{"CHARACTER"})$. 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 alгореf`

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 routineexpr exists, $\text{\textasciitilde}\$ROUTINE(\text{routineexpr})$ has a value; all nonempty string values are reserved for future enhancement of the International Standard. Process characteristics are stored beneath the $\text{\textasciitilde}\$ROUTINE(\text{routineexpr})$ node:

$\text{\textasciitilde}\$ROUTINE(\text{ routineexpr } , \text{ expr } \vee \text{ "CHARACTER" }) = \text{ charsetexpr }$

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

When a routine is created and $\text{\textasciitilde}\$ROUTINE(\text{routineexpr}, \text{ "CHARACTER" })$ for that routine has a \$DATA value of zero, then this node is assigned the current value of the node $\text{\textasciitilde}\$JOB(\$JOB, \text{ "CHARACTER" })$.

7.1.3.7 $\text{\textasciitilde}\$SYSTEM$

$\text{\textasciitilde}\$[SYSTEM] (\text{ systemexpr })$

systemexpr ::= expr $\vee$ system

system ::= syntax of \$SYSTEM intrinsic special variable

$\text{\textasciitilde}\$SYSTEM$ 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 $\text{\textasciitilde}\$SYSTEM$ not beginning with the letter "Z" are reserved for future enhancement of the International Standard.

System Character Set Profile:

$\text{\textasciitilde}\$SYSTEM(\text{ systemexpr } , \text{ expr } \vee \text{ "CHARACTER" }) = \text{ charsetexpr }$

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 $\text{\textasciitilde}\$Z[\text{unspecified}]$

$\text{\textasciitilde}\$Z[\text{unspecified}] (\text{ unspecified })$

will provide implementation-specific information. Z is the initial letter for defining non-standard structured system variables. The requirement that $\text{\textasciitilde}\$Z$ 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:

$\text{\textasciitilde}\$JOB(\text{processid}, \text{ "GLOBAL" })$	default global environment
$\text{\textasciitilde}\$JOB(\text{processid}, \text{ "LOCK" })$	default lock environment
$\text{\textasciitilde}\$JOB(\text{processid}, \text{ "ROUTINE" })$	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

<u>expritem</u>	::=	<u>strlit</u> <u>numlit</u> <u>exfunc</u> <u>exvar</u> <u>svn</u> <u>function</u> <u>unaryop</u> <u>expratom</u> (<u>expr</u>)
-----------------	-----	---

See 7.1.5 for a definition of function.

7.1.4.1 String literal strlit

<u>strlit</u>	::=	"	" " <u>nonquote</u>	... "
---------------	-----	---	------------------------	-------

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, intlitt, which is a nonempty string of digits, is defined here.

$$\text{intlitt} ::= \text{digit} \dots$$

The numeric literal numlit is defined as follows.

$$\text{numlit} ::= \text{mant} [\text{exp}]$$

$$\text{mant} ::= \left| \begin{array}{l} \text{intlitt} [\text{. intlitt}] \\ \text{. intlitt} \end{array} \right|$$

$$\text{exp} ::= E \left[\begin{array}{c} + \\ - \end{array} \right] \text{intlitt}$$

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: $+ T \Rightarrow T$)
- b) $- + T \Rightarrow - T$
- c) $-- T \Rightarrow T$

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