1984

a100002: Title Page

ANSI
X11.1–1984

American National Standard

for information systems --
programming language --
MUMPS

Sponsor
MUMPS Development Committee

Approved November 15, 1984
American National Standards Institute, Inc.

Abstract

This standard contains a three-part description of various aspects of the MUMPS computer programming language. Part I, the MUMPS Language Specification, consists of a stylized English narrative definition of the MUMPS language. Part II, the MUMPS Transition Diagrams, represents a formal definition of the language described in Part I, employing a form of line drawings to illustrate syntactic and semantic rules governing each of the language elements. Part III, the MUMPS Portability Requirements, identifies constraints on the implementation and use of the language for the benefit of parties interested in achieving MUMPS application code portability.

a100003: Committee Statements

American National Standard

Approval of an American National Standard requires verification by ANSI that the requirements for due process, consensus, and other criteria for approval have been met by the standards developer.

Consensus is established when, in the judgment of the ANSI Board of Standards Review, substantial agreement has been reached by directly and materially affected interests. Substantial agreement means much more than a simple majority, but not necessarily unanimity. Consensus requires that all views and objections be considered, and that a concerted effort be made toward their resolution.

The use of American National Standards is completely voluntary: their existence does not in any respect preclude anyone, whether he has approved the standards or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standards.

CAUTION NOTICE: This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken to reaffirm, revise, or withdraw this standard no later than five years from the date of approval. Purchasers of American National Standards may receive current information on all standards by calling or writing the American National Standards Institute.

Published by MUMPS Development Committee
c/o Richard F. Walters. Ph.D.
Division of Computer Science
University of California
Davis, California 95616

the prior written permission of the publisher.

Printed in the United States of America

Library of Congress Catalog Card Number: 85–061202
ISBN 0–918118–30–1

a100004: Foreword

Foreword

(This Foreword is not a part of American National Standard ANSI AMC   1984.)

MUMPS, an acronym for Massachusetts General Hospital Utility Multi-Programming System. is a high level-interactive computer programming language developed for use in complex data handling operations. The MUMPS Development Committee has accepted responsibility for creation and maintenance of the language since early 1973. The first ANSI approved standard was approved Sept. 15, 1977 via the canvass method. Subsequently, the MUMPS Development Committee has met several times annually to consider revisions to the standard. The new revisions were submitted to ANSI for processing via the canvass method Oct. 15, 1983. Approval was granted on November 15, 1984.

Document preparation was performed by the MUMPS Development Committee. Suggestions for improvement of this standard are welcome. They should be submitted to the MUMPS Development Committee, c/o Richard F. Walters. Division of Computer Science. University of California, Davis. CA 95616.

Consensus for approval if this standard as an American National Standard was achieved through the use of the Canvass Method. The following organizations responded to the canvass.

Organization	Representative
Ackerman Business Consultants	Thomas G. Ackerman
American Management Systems	Frank Kirschenbaum
Association of Computer Programmers and Analysts	Lawrence Ruh
Baylor College of Medicine	Mohan S. Beltangady
Comp Consultants	David B. Brown
Computer Industry Associates	N.J. Ream
Computer People	Charles Volkstorf
Control Data Corporation	Charles E. Cooper
Data Processing Management Association	Ardyn E. Dubnow
Data Tree, Inc.	Peter Beaman
Department of Defense	Michael Todd
Digital Equipment Corporation	Michael Banovsky John Cassell
Exec-U-Comp Information System	Harry S. White
George H. Gardner, III	George H. Gardner
George Washington University	Helmuth F. Orthner
Harborview Hospital	Arden Forrey
Harvard Comm. Health	Lawrence Goldstein
Honeywell	Lance J. Miller Peter J. Singer
Innovative Computer Systems	Jay Cruce
Interactive Software Products	Martin Johnson
Intersystems, Inc.	Terrance Ragon
Ithaca College	Susan Bailey
Johns Hopkins University	Thomas C. Salander
Massachusetts General Hospital	G. Octo Barnett Dan Souder
Micronetics	David Marcus Richard T. May

Mitre Corporation	Barbara Jacobs Benjamin C. Locke
Mt. Sinai Hospital	Jonathan Trask
MUMPS Etcetera	Fred Davidson
National Center Health Service Research	Donald Barnes
National Computer Systems	Bob Isch
National Library of Medicine	Robert Williamson
Ohio State University	Jack W. Smith
Oklahoma City Pediatrics Hospital	Thomas Harris
Patterson, Gray Associates	Mark Patterson R. Marcus Gray
Pearson, Donahue Systems	J.M. Pearson
Sakowitz Computer Lab	Donald H. Glaeser
Shared Medical Systems	Charles Drum Brian Lucas
Sperry Corporation	C.D. Card
Stanford University	Terrence Moore
Bryan Stone	Bryan Stone
Think Technologies	Melvin V. Conway
University of California, Davis	Richard F. Walters
University of California, San Francisco	Mark Shafarman
University of Massachusetts Medical Center	Kathleen F. McCarthy
University of Regina	Robert Greenfield
University of Tennessee	Fletcher Donaldson Kevin O'Kane Arthur Kretchman
Veterans Administration	Marina Bates Carl Bauer Richard Davis Charles F. Leahey Robert Lushene Thomas Munnecke George Timson
Vista Computers	Kevin O'Gorman
Kirt Walker Assoc.	Kirt Walker

a101002: Document Organization

1. Overview of MUMPS Language Specification

1.1 Organization of this Document

This document describes the MUMPS language at two levels of detail. Subsection 1.2 gives an overview of the prominent features of the language, intended for the reader who is already familiar with at least one existing dialect. Section 3 describes the static syntax of the language. The distinction between "static" and "dynamic" syntax is as follows. The static syntax describes the sequence of characters in a program as it appears on a tape in program interchange or on a listing. The dynamic syntax describes the sequence of characters actually encountered by an interpreter during execution of the program. The dynamic syntax takes into account transfers of control and values produced by indirection. Section 2 describes the metalanguage used for the static syntax.

a101003: Character Set

1.2 Summary of the Language

1.2.1 Character Set

The character set which is used for the interchange of MUMPS programs and data is the seven-bit USA Standard Code for Information Interchange (ASCII) defined by ANSI X3.4–1977. Programs may be written entirely with the common 64-character subset of ASCII. The character collating sequence is the same as the numeric sequence of the ASCII character codes.

a101004: Routine Structure

1.2.2 Routine Structure

A MUMPS routine consists of a sequence of lines. For purposes of transfer of control, lines may be optionally labeled. A label is either a conventional MUMPS name (an initial letter or % followed by alphanumerics), or it is an integer literal.

---

a101005: Program Punctuation

1.2.3 Program Punctuation

The following special characters may occur in programs.

Unary Arithmetic Operators

+	plus
-	negate

Unary Logical Operator

'	not

Binary Arithmetic Operators

+	addition
-	subtraction
*	multiplication
/	division
\	division with integer quotient
#	modulo

Binary Relational Operators

<	numeric less than
'<	numeric greater than or equal
>	numeric greater than
'>	numeric less than or equal
=	string identity
'=	string non-identity
[	string contains
'[	string non-contains
]	string follows
']	string non-follows
?	string pattern match
'?	string pattern non-match

Binary Logical Operators

&	and
'&	nand
!	or
'!	nor

Binary String Operator

_	concatenation

Delimiters

,	argument separation, subscript separation
=	value assignment
:	post-conditional expression, subargument separation
( )	grouping
@	indirection
"	string literals
.	decimal point in numeric literals
^	preceding routine name in DO, GOTO
E	preceding exponent in numeric literals
;	comment
space	separating command words

Prefixes

^	global variable names
$	functions, special variable names
%	available in names of the programmer's choice

a101006: Data Types

1.2.4 Data Types

Arithmetic operations are performed on strings and produce numeric values, which are special cases of strings. This approach to the standard specification does not preclude the use of multiple data representations within an implementation of the standard.

Any string value may enter into an arithmetic operation; there is a uniform rule for interpreting a string as a number. Certain operations deal with integer values, which are special cases of numeric values; the latter may contain decimal fractions. There is a uniform rule for interpreting any number (and, by inference, any string) as an integer.

Certain other operations deal with truth values, which are special cases of numeric values. There are two truth values: 0 and 1. The integer value 0 is the truth value 0. The integer value 1 is the truth value 1. All other numeric values are interpreted as the truth value 1. The truth value 0 denotes False; the truth value 1 denotes True.

a101007: Precedence of Operators

1.2.5 Precedence of Operators

All binary operators are at the same level of precedence. Application of unary operators precedes application of binary operators.

a101008: Commands

1.2.6 Commands

The following commands are defined.

BREAK	provides an access point within the standard for non-standard programming and debugging aids.
CLOSE	releases one or more devices from ownership.
DO	provides a generalized subroutine call.
ELSE	permits conditional execution.
FOR	controls repetitive execution over a set of values of a variable.
GOTO	provides a generalized transfer of control.
HALT	terminates execution.
HANG	suspends execution for a specified period of time.
IF	permits conditional execution.

JOB	initiates a new MUMPS process.
KILL	controls the elimination of specified variables and their values.
LOCK	provides a generalized interlock facility for coordinating concurrent processes.
OPEN	obtains ownership of one or more devices.
QUIT	defines an exit point of FOR or DO.
READ	specifies data input.
SET	assigns values to variables.
USE	designates a specific device for input and output.
VIEW	provides an access point within the standard for the examination of machine-dependent information.
WRITE	specifies data output.
XECUTE	permits execution of strings arising from the expression evaluation process.
Z	reserved for implementation-specific extensions.

All other command words, except those beginning with Z, are reserved.

a101009: Functions

1.2.7 Functions

The following functions are currently specified.

$ASCII	selects a character of a string and returns its code as an integer.
$CHAR	translates a set of integers into a string of characters whose codes are those integers.
$DATA	returns an integer specifying whether a defined value and/or pointer of a named variable exists.
$EXTRACT	returns a character or substring of a string expression, selected by position number.
$FIND	returns an integer specifying the end position of a specified substring within a string.
$JUSTIFY	returns the value of an expression, right-justified within a field of specified size.
$LENGTH	returns the length of a string or the number of occurrences of a specified substring within a specified string.
$NEXT and $ORDER	return the lowest numeric subscript value on the same level, but numerically higher than the last subscript of the named global or local variable.
$PIECE	returns a string between two specified occurrences of a specified substring within a specified string.
$RANDOM	returns a pseudo-random number in a specified interval.
$SELECT	returns the value of one of several expressions in a list, selected by the truth values in a second list of expressions.
$TEXT	returns the text content of a specified line of the routine in which the function appears.
$VIEW	reserved for implementation-specific methods of obtaining machine-dependent data.
$Z	reserved for definition of implementation-specific functions.

All other initial letters of function names are reserved.

a101010: Special Variables

1.2.8 Special Variables

The following special variables are specified.

$HOROLOG	provides the date and time in a single, two-part value.
$IO	identifies the currently assigned I/O device.
$JOB	has an integer value which uniquely identifies the process which evaluates it.
$STORAGE	provides the number of unused characters which remain in a routine's partition.
$TEST	makes available the truth value determined by the IF command and by the OPEN, LOCK, and READ with timeouts.
$X	gives the horizontal cursor position on the current device.
$Y	gives the line number on the current device.

$Z reserved for implementation-specific definitions. All other initial letters of special variable names are reserved.

a105001: Metalanguage Description

2. Static Syntax Metalanguage

The primitives of the metalanguage are the ASCII characters and the metalanguage operators (definition), [ ] (option), ; ; (grouping), ... (optional indefinite repetition), L (list), and V (value).

In general, defined syntactic objects will have designations 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 [1, and three dots ... denote that the previous element is optionally repeated any number of times. The' definition of name, for example, is written:

$$\underline{name} ::= \left| \begin{matrix} \underline{\%} \\ \underline{alpha} \end{matrix} \right| \left[\begin{matrix} \underline{digit} \\ \underline{alpha} \end{matrix} \right] \ldots$$

The vertical bars are used only to group elements for repetition or to make a group of elements more readable. When there is any danger of confusing the square brackets in the metalanguage with the ASCII graphics [and], special care is taken to avoid this. 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,

$$\underline{L}\ \underline{name}$$

is equivalent to

$$\underline{name}\ [\ ,\ \underline{name}\ ]\ \ldots$$

The binary metalanguage operator V, used in the specification of indirection, places the constraint on the expratom 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

$$\underline{examplecommand} ::= \text{EXAMPLE}\ \text{Space}\ \underline{L}\ \underline{exampleargument}$$

where

$$\underline{exampleargument} ::= \left| \begin{matrix} \underline{expr} \\ @\ \underline{expratom}\ \underline{V}\ \underline{L}\ \underline{exampleargument} \end{matrix} \right|$$

This says that, after evaluation of indirection, the command argument list consists of any number of exprs separated by commas. In the static syxtax (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.

a105002: Basic Alphabet

3. Static Syntax
3.1 Basic Alphabet

The routine, which is the object whose static syntax is being described in Section 3, is a string made up of the following 98 symbols.

- The 95 ASCII printable characters, including SP (space)
- The line-start symbol ls
- The end-of-line symbol eol
- The end-of-routine symbol eor

In program interchange, the following ASCII characters are used in place of ls, eol, and eor.

- ls: SP
  eol: CR LF
  eor: CR FF

When a program is stored internally, the standard does not specify what forms ls, eol, and eor take. They may, in fact, be expressed by means other than characters in the program. When a program is entered from a keyboard, the standard does not specify what operator procedures correspond to ls, eol, or eor.

The syntactic types graphic, alpha, and digit are defined here informally in order to save space.

graphic ::= any of the class of 95 ASCII printable characters, including SP (space), represented by Space or SP.

alpha ::= any of the class of 52 upper and lower case letters: A-Z, a-z.

digit ::= any of the class of 10 digits: 0–9.

a106001: routine

3.4 Routines

The routine is the unit of program interchange. In program interchange, each routine begins with its routinehead, which contains the identifying routinename, and the routinehead is followed by the routinebody, which contains the executed code. The routinehead is not part of the executed code.

routine ::= routinehead routinebody

See also the transition diagram for routine.

routinehead ::= routinename eol

routinename ::= name

The routinebody is a sequence of lines terminated by the eor. Each line ends with eol, starts with ls optionally preceded by a label, and may contain zero or more commands (separated by one or more spaces) between ls and eol. Any line may end with a comment immediately preceding the eol. Zero or more spaces may separate the comment from the last command of the line.

routinebody ::= line [line] ... eor

$$\text{line ::= [label] ls} \left[\begin{array}{c} \text{commands [[cs] comment]} \\ \text{[cs] comment} \end{array} \right] \text{eol}$$

See also the transition diagram for line.

$$\text{label ::=} \left| \begin{array}{c} \text{name} \\ \text{intlit} \end{array} \right|$$

See also the transition diagram for label.

ls ::= SP [SP] ...   (one or more spaces)

cs ::= SP [SP] ...   (one or more spaces)

eol ::= CR LF   (two control characters)

eor ::= CR FF   (two control characters)

Each occurrence of a label to the left of ls in a line is called a "defining occurrence" of label. No two defining occurrences of label may have the same spelling in one routinebody.

a106002: routinehead

3.2.1 Name name

```
name ::= | %     | [ digit ] ...
         | alpha | [ alpha ]
```

See also the transition diagram for name.

a107001: expr

3.3 Expressions expr

Expressions are made up of expression atoms separated by binary string, arithmetic, or truth-valued operators.

```
expr ::= expratom [ exprtail ] ...
```

See also the transition diagram for expr.

```
            | |    binaryop     |          |
exprtail ::=| | [ ' ] truthop   | expratom |
            |                              |
            |      [ ' ] ? pattern         |

binaryop ::= | _ |     (Note: underscore)
             | + |
             | - |     (Note: hyphen)
             | * |
             | / |
             | \ |
             | # |
```

See also the transition diagram for binaryop.

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

See also the transition diagram for truthop.

```
              =
relation ::= | [ |
             | < |
             | ] |
              >

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

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.

a107002: expratom

3.2 Expression Atom expratom

The expression, expr, is the syntactic element which denotes the execution of a value-producing calculation; it is defined in 3.3. The expression atom, expratom, is the basic value-denoting object of which expressions are built; it is defined here.

```
             |   lvn    |
expratom ::= |   gvn    |
             | expritem |
```

See also the transition diagram for expratom.

```
             |        svn         |
             |      function      |
expritem ::= |       numlit       |
             |       strlit       |
             |      ( expr )      |
             | unaryop expratom   |
```

```
            | ' |      (Note: apostrophe)
unaryop ::= | + |
            | - |      (Note: hyphen)
```

See also the transition diagram for unaryop.

a107006: Variables

3.2.2 Variables

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

A MUMPS 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.

MUMPS 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 available to all routines executed within the processes lifetime. Although the concept of limiting the environment of certain local variables to given routines or subroutine domains is under discussion for inclusion in the MUMPS language, no such specification is included as a part of this revision of the MUMPS Standard.

A Global variable is one that is created by a MUMPS process, but is permanent and shared. As soon as it has been created, it is accessible to other MUMPS processes on the system. Global variables do not disappear when a process terminates. Like local variables, global variables are available to all routines executed within a process.

The two basic differences between MUMPS definitions of local and global variables are: 1) there is no implied restriction of environment within a process attached to either local or global variables; and 2) global variables are immediately available to be shared with other processes active on a MUMPS system, whereas local variables cannot be shared with other processes on the same system.

a107007: glvn

3.2.2.2 Global Variable Name gvn

For convenience, glvn is defined so as to be satisfied by the syntax of either gvn or lvn.

```
         | gvn |
glvn ::= | lvn |
```

| |

See also the transition diagram for glvn.

a107008: lvn

3.2.2.1 Local Variable Name lvn

lvn ::= | rlvn |
| @ expratom V> lvn |

See also the transition diagram for lvn.

rlvn ::= | name [(L expr)] |
| @ lnamind @ (L expr) |

lnamind ::= rexpratom V lvn

rexpratom ::= | rlvn |
| rgvn |
| 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.

A lnamind 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.

a107011: gvn

3.2.2.2 Global Variable Name gvn

gvn ::= | rgvn |
| @ expratom V gvn |

See also the transition diagram for gvn.

rgvn ::= | ^ (L expr) |
| ^ name [(L expr)] |
| @ gnamind @ (L expr) |

gnamind ::= rexpratom V gvn

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. There is permitted an abbreviated form of subscripted gvn, called the "naked reference", in which the 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.

A gnamind 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 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, ..., v_{m-1}, is placed into the naked indicator when the gvn reference is made. A subsequent naked reference of the form (s_1, s_2, ..., s_i) (i positive) results in a global reference of the form N(v_1, v_2, ..., v_{m-1}, s_1, s_2, ..., s_i) after which the m+i–1-tuple N, v_1, v_2, ..., 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. It is erroneous for the first executed occurrence of gvn to be a naked reference. A nonnaked reference without subscripts leaves the naked indicator undefined.

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.

---

a107048: strlit

3.2.6 String Literal strlit

Let nonquote temporarily be defined as any of the class of 94 graphics, excluding the quote symbol.

$$\text{strlit} ::= \text{"} \left[\begin{matrix}\text{""} \\ \text{nonquote}\end{matrix}\right] \ldots \text{"}$$

See also the transition diagram for strlit.

In words, a string literal is bounded by quotes and contains any string of printable characters, except that when quotes occur inside, 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.

---

a107049: numlit

3.2.4 Numeric Literal numlit

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

$$\text{intlit} ::= \text{digit} \; [\; \text{digit} \;] \ldots$$

See also the transition diagram for intlit.

The numeric literal numlit is defined as follows.

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

See also the transition diagram for numlit.

$$\text{mant} ::= \left|\begin{matrix}\text{intlit} \; [\; .\; \text{intlit} \;] \\ .\; \text{intlit}\end{matrix}\right|$$

$$\text{exp} ::= \text{E} \left[\begin{matrix} + \\ - \end{matrix}\right] \text{intlit}$$

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

---

a107050: Numeric Data Values

3.2.4.1 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 may 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.

---

a107051: Meaning of numlit

3.2.4.2 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".

---

a107052: Numeric Interpretation

3.2.5 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 all of the characters of the string to the left of a given point and none of the characters of 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 $_{\text{Right Arrow}}$ T)
b. - + T $_{\text{Right Arrow}}$ - T
c. - - T $_{\text{Right Arrow}}$ 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 3.2.4.2 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

See also the transition diagram for numexpr.

---

a107053: Integer Interpretation

3.2.5.1 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

See also the transition diagram for intexpr.

---

a107054: Truth-Value Interpretation

3.2.5.2 Truth-Value Interpretation

The truth-value interpretation is a mapping from the set of all data values onto the two integer values 0 and 1, 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

See also the transition diagram for tvexpr.

---

a107057: svn

3.2.7 Special Variable Name svn

Special variables are denoted by the prefix $ followed by one of a designated list of names. Special variable names differing only in the use of corresponding upper and lower case letters are equivalent. Any of the following defined special variables satisfies the definition of svn.

$H[OROLOG]
$I[O]
$J[OB]
$S[TORAGE]
$T[EST]
$X
$Y
$Z[unspecified]

---

a107062: $HOROLOG

3.2.7 Special Variable Name svn

Syntax	Definition
$H[OROLOG]	$H 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 $H 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.

See also the transition diagram for $Horolog.

---

a107063: $IO

3.2.7 Special Variable Name svn

Syntax	Definition
$I[O]	$I identifies the current I/O device. See 3.6.2 and 3.6.16.

See also the transition diagram for $IO.

---

a107065: $JOB

3.2.7 Special Variable Name svn

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

See also the transition diagram for $Job.

---

a107072: $STORAGE

3.2.7 Special Variable Name svn

Syntax	Definition
$S[TORAGE]	Each implementation must return for the value of $S an integer which is the number of characters of free space available for use. The method of arriving at the value of $S is not part of the standard.

See also the transition diagram for $Storage.

---

a107074: $TEST

3.2.7 Special Variable Name svn

Syntax	Definition
$T[EST]	$T contains the truth value computed from the execution of the most recent IF command containing an argument, or an OPEN, LOCK, or READ with a timeout.

See also the transition diagram for $Test.

---

a107077: $X

3.2.7 Special Variable Name svn

Syntax	Definition

$X	$X has a nonnegative integer value which approximates the value of a carriage or horizontal cursor position on the current line as if the current I/O device were an ASCII terminal. It is initialized to zero by input or output of control functions corresponding to CR or FF; input or output of each graphic adds 1 to $X. See 3.5.5 and 3.6.16.

See also the transition diagram for $X.

a107078: $Y

3.2.7 Special Variable Name svn

Syntax	Definition
$Y	$Y has a nonnegative integer value which approximates the line number on the current I/O device as if it were an ASCII terminal. It is initialized to zero by input or output of control functions corresponding to FF; input or output of control functions corresponding to LF adds 1 to $Y. See 3.5.5 and 3.6.16.

See also the transition diagram for $Y.

a107079: $Z*

3.2.7 Special Variable Name svn

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

a107080: unaryop

3.2.9 Unary Operator unaryop

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.

a107082: Intrinsic Function

3.2.8 Functions function

Functions are denoted by the prefix $ followed by one of a designated list of names, followed by a parenthesized argument list. Function names differing only in the use of corresponding upper and lower case letters are equivalent. Any of the following specifications satisfies the definition of function.

$A[SCII]
$C[HAR]
$D[ATA]
$E[XTRACT]
$F[IND]
$J[USTIFY]
$L[ENGTH]

$N[EXT]
$O[RDER]
$P[IECE]
$R[ANDOM]
$S[ELECT]
$T[EXT]
$V[IEW]
$Z[unspecified]

a107084: $ASCII

3.2.8 Functions function

$A[SCII](expr)

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 $A($C(n)) = n.

$A[SCII](expr$_1$, intexpr$_2$)

is similar to $A(expr$_1$) except that it works with the intexpr2th character of expr$_1$ instead of the first. Formally, $A(expr$_1$,intexpr$_2$) is defined to be $A($E(expr$_1$,intexpr$_2$)).

See also the transition diagram for $ASCII.

a107085: $CHAR

3.2.8 Functions function

$C[HAR](L intexpr)

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 $C.

See also the transition diagram for $Char.

a107086: $DATA

3.2.8 Functions function

$D[ATA](glvn)

returns a nonnegative integer which is a characterization of the variable named. The value of the integer is p+d, where:

- d = 1 if the named variable has a defined value (that is, it exists); d = 0 otherwise;
- p = 10 if either:
  a. The named variable contains no subscripts, and there exists a subscripted variable with the same name, or
  b. The named variable contains n subscripts, and there exists a subscripted variable with m > n subscripts whose first n subscript values are the same as the values of those in the named variable;
- p = 0 otherwise.

See also the transition diagram for $Data.

a107089: $EXTRACT

3.2.8 Functions function

$E[XTRACT](expr)

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

$E[XTRACT](expr$_1$, intexpr$_2$)

Let s be the value of expr$_1$, and let m be the integer value of intexpr$_2$. $E(s,m) returns the mth character of s. If m is less than 1 or greater than $L(s), the value of $E is the empty string. (1 corresponds to the leftmost character of s; $L(s) corresponds to the rightmost character.)

$E[XTRACT](expr$_1$, intexpr$_2$, intexpr$_3$)

Let n be the integer value of intexpr$_3$. $E(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 Section 3.3.5, $E(s,m,n) _ $E(s,m) $E(s,m+1,n).
d. m < n and $L(s) < n. $E(s,m,n) $E(s,m,$L(s)).

See also the transition diagram for $Extract.

---

a107090: $FIND

3.2.8 Functions function

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

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

$F[IND] (expr$_1$, expr$_2$, intexpr3)

Let a be the value of expr$_1$, let b be the value of expr$_2$, and let m be the value of intexpr$_3$. $F(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 $F($E(a,m,$L(a)),b). If no instance of b is found (i.e., p=0), $F returns the value O; otherwise, $F(a,b,m) = p + max(m,1) - 1.

See also the transition diagram for $Find.

---

a107094: $JUSTIFY

3.2.8 Functions function

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

returns the value of expr$_1$ right-justified in a field of intexpr$_2$ spaces. Let m be $L(expr$_1$) and n be the value of intexpr$_2$. The following cases are defined:

a. m > n. Then the value returned is expr$_1$.
b. Otherwise, the value returned is S(n-m) concatenated with expr$_1$, where S(x) is a string of x spaces.

$J[USTIFY](numexpr$_1$, intexpr$_2$, intexpr$_3$)

returns an edited form of the number numexpr$_1$. Let r be the value of numexpr$_1$ after rounding to intexpr$_3$ fraction digits, including possible trailing zeros. (If intexpr$_3$ is the value 0, r contains no decimal point.) The value returned is $J(r, intexpr2). Note that if the value of numexpr$_1$ is between –1 and 1, the result of $J does have a zero to the left of the decimal point. Negative values of intexpr$_3$ are reserved for future extensions of the $JUSTIFY function.

See also the transition diagram for $Justify.

---

a107095: $LENGTH

3.2.8 Functions function

$L[ENGTH](expr)

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

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

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 $L returns the value 0.

See also the transition diagram for $Length.

---

a107099: $NEXT

3.2.8 Functions function

$N[EXT](glvn)

is included for backward compatibility. The use of the $ORDER function is strongly encouraged in place of $NEXT, as the two functions perform the same operation except for the different starting and ending condition of $NEXT. $N 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, CO(s,t) returns t. Otherwise, CO(s,t) returns s.

Let m and n be strings satisfying the definition of numeric data values (Section 3.2.4.1), and u and v be nonempty strings which do not satisfy this definition. The following cases define the ordering sequence:

a. CO("",s) = s.
b. CO(m,n) = n if n > m; otherwise, CO(m,n) = m.
c. CO(m,u) = u.
d. CO(u,v) = v if v] u; otherwise, CO(u,v) = u.

In words, all strings follow the empty string, numerics collate in numeric order, numerics precede nonnumeric strings, and nonnumeric strings are ordered by the conventional ASCII collating sequence.

Only subscripted forms of lvn and gvn are permitted. Let lvn or gvn be of the form Name(s_1, s_2, ..., s_n). If s_n is –1, let A be the set of all subscripts. If s_n is not –1, let A be the set of all 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 $N(Name($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, –1 is returned.

Note that $N will return ambiguous results for lvn and gvn arrays which have negative numeric subscript values.

See also the transition diagram for $Next.

a107100: $ORDER

3.2.8 Functions function

$O[RDER](glvn)

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

Let m and n be strings satisfying the definition of numeric data values (Section 3.2.4.1), and u and v be nonempty strings which do not satisfy this definition. The following cases define the ordering sequence:

a. CO("",s) = s.
b. CO(m,n) = n if n > m; otherwise, CO(m,n) = in.
c. CO(m,u) = u.
d. CO(u,v) = v if v] u; otherwise, CO(u,v) = u.

In words, all strings follow the empty string, numerics collate in numeric order, numerics precede nonnumeric strings, and nonnumeric strings are ordered by the conventional ASCII collating sequence.

Only subscripted forms of lvn and gvn are permitted. Let lvn or gvn 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(sn,s) = s and
b. $D(Name($s_1$, s_2, ..., $s_{n–1}$, s)) is not zero.

Then $O(Name($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, $O returns the empty string.

See also the transition diagram for $Order.

a107101: $PIECE

3.2.8 Functions function

$P[IECE] (expr$_1$, expr$_2$)

is defined here with the aid of a function, NF, which is used for definitional purposes only, called "find the position number following the mth occurrence".

NF(s,d,m) is defined, for strings s, d, and integer in, 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 generalizes $F 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 expr$_1$, and let d be the value of expr$_2$. $P(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,O,NF(s,d,1) - $L(d) - 1).

$P[IECE](expr$_1$, expr$_2$, intexpr$_3$)

Let m be the integer value of intexpr$_3$. $P(s,d,m) returns the substring of s bounded by but not including the m-lth and the mth occurrence of d.

$P(s,d,m) _ $E(s,NF(s,d,m–1),NF(s,d,m) - $L(d) - 1).

$P[IECE](expr$_1$, expr$_2$, intexpr$_3$, intexpr$_4$)

Let n be the integer value of intexpr$_4$. $P(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).

See also the transition diagram for $Piece.

---

a107105: $RANDOM

3.2.8 Functions function

$R[ANDOM] (intexpr)

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 will occur.

See also the transition diagram for $Random.

---

a107107: $SELECT

3.2.8 Functions function

$S[ELECT] (L | tvexpr : expr |)

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 $S. An error will occur if all tvexprs are false. Since only one expr is evaluated at any invocation of $S, that is the only expr which must have a defined value.

See also the transition diagram for $Select.

---

a107109: $TEXT

3.2.8 Functions function

$T[EXT] ($\left|\begin{array}{c} + \text{intexpr} \\ \text{lineref} \end{array}\right|$)

returns a string whose value is the contents of the line in this routine specified by the argument. Specifically, the entire line, with ls replaced by one SP and eol deleted, is returned.

If the argument of $T is a lineref, the line denoted by the lineref is specified. If the argument is + intexpr, 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 will occur if the value of intexpr is less than 0.

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.

See also the transition diagram for $Text.

---

a107112: $VIEW

3.2.8 Functions function

$V[IEW] (unspecified)

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

---

a107113: $Z*

3.2.8 Functions function

$Z[unspecified] (unspecified)

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

---

a107192: Concatenation

3.3.5 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.

---

a107193: Arithmetic Operators

3.3.1 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 argument is positive and one argument is negative. Division by zero is erroneous.
\	produces the integer interpretation of the result of the above division.
#	produces the value of the left argument modulo the right argument. It is defined only for nonzero values of its right argument, as follows. $A \# B = A - (B * \mathrm{floor}(A/B))$ where floor (x) = the largest integer < x.

---

a107195: relation

3.3.2 Relational Operators

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

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

---

a107196: Numeric Relations

3.3.2.1 Numeric Relations

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

---

a107197: String Relations

3.3.2.2 String Relations

The relations =] [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 " $F(A,B). Note that the emply 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 conventional ASCII collating 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 code 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., $E(A,1,n) $E(B,1,n)) and the remainder of A follows the remainder of B (i .e . , $E(A,n+1, $L(A)) follows $E(B,n+1,$L(B))).

---

a107198: logicalop

3.3.4 Logical Operators

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.

$$A\ !\ B = \begin{cases} 0 \text{ if both A and B have the value 0} \\ 1 \text{ otherwise} \end{cases}$$

$$A\ \&\ B = \begin{cases} 1 \text{ if both A and B have the value 1} \\ 0 \text{ otherwise} \end{cases}$$

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

A '& B = '(A & B)
A '! B = '(A! B)

---

a107199: pattern

3.3.3 Pattern match

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.

```
pattern ::= | patatom [ patatom ] ... |
            | @ expratom V pattern    |
```

See also the transition diagram for pattern.

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

$$S = S_1\ S_2 \ldots S_n$$

such that there is a one-to-one order-preserving correspondence between the Si and the pattern atoms, and each

Si "satisfies" its respective pattern atom. Note that some of the Si may be empty.

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

$$\text{patatom} ::= \text{repcount} \left| \begin{array}{c} \text{patcode} \\ \text{strlit} \end{array} \right|$$

See also the transition diagram for patatom.

$$\text{repcount} ::= \left| \begin{array}{c} \text{intlit} \\ [\ \text{intlit}_1\ ]\ .\ [\ \text{intlit}_2\ ] \end{array} \right|$$

$$\text{patcode} ::= \left| \begin{array}{c} \text{C} \\ \text{N} \\ \text{P} \\ \text{A} \\ \text{L} \\ \text{U} \\ \text{E} \end{array} \right| \ldots$$

See also the transition diagram for patcode.

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, as follows.

C	33 ASCII control characters, including DEL
N	10 ASCII numeric characters
P	33 ASCII punctuation characters, including SP
A	52 ASCII alphabetic characters
L	26 ASCII lower-case alphabetic characters
U	26 ASCII upper-case alphabetic characters
E	Everything (the entire set of ASCII characters) All other unused patcode letters for class names are reserved. 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 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, intlitl.intlit$_2$, the first intlit gives the lower bound, and the second intlit the upper bound. It is erroneous if the upper bound is less than the lower bound. 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 intlitl. , the upper bound is taken to be indefinite; that is, the range is at least intlitl occurrences. Then patatom is satisfied by the concatenation of a number of Si, each of which meets the specification of patatom, where the number must be within the expressed or implied bounds of the specified range, inclusive.

The dual operator '? is defined by: A '? B = '(A ? B)

---

a108001: General Command Rules

3.5 General command Rules

Every command starts with a "command word" which dictates the syxtax and interpretation of that command instance. Command words differing only in the use of corresponding upper and lower case letters are equivalent. The standard contains the following command words.

B[REAK]
C[LOSE]
D[O]
E[LSE]
F[OR]
G[OTO]
H[ALT]
H[ANG]
I[F]
J[OB]
K[ILL]
L[OCK]
O[PEN]
Q[UIT]
R[EAD]
S[ET]
U[SE]
V[IEW]
W[RITE]
X[ECUTE]
Z[unspecified]

Unused initial letters of command words are reserved for future enhancement of the standard.

The formal definition of the syntax of command is a choice from among all of the individual command syntax definitions of 3.6.

$$\text{command} ::= \left| \begin{array}{c} \text{syntax of the BREAK command} \\ \text{syntax of the CLOSE command} \\ \ldots \\ \text{syntax of the XECUTE command} \end{array} \right|$$

See also the transition diagram for command.

Any implementation of the language must be able to recognize both the initial letter abbreviation and the full spelling of each command word. When two command words have a common initial letter, their argument syntaxes uniquely distinguish them.

For all commands allowing multiple arguments, the form

command word arg_1, arg_2 ...

is equivalent in execution to

command word arg_1 command word arg_2 ...

---

a108002: Spaces in Commands

3.5.2 Spaces in Commands

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

a. There may be a SP immediately preceding eol only if the line ends with a comment. (Since ls may immediately precede eol, this rule does not apply to the SP which may stand for ls.)
b. If a command instance contains at least one argument, the command word 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 command word or postcond is followed by at least two spaces; if it is the last command of the line and no comment follows, the command word or postcond is immediately followed by eol.

---

a108003: comment

3.5.3 Comments

If a semicolon appears in the command word 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.

comment ::= ; [graphic] ...

---

a108004: Command Argument Indirection

3.5.8 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 command word carries an argument list, as in

COMMANDWORD , , L argument

the argument syntax will be expressed as

```
             | individual argument syntax |
argument ::= |  @ expratom V L argument   |
```

See also the transition diagram for 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.

---

a108005: postcond

3.5.1 Post Conditionals

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

postcond ::= [: tvexpr]

See also the transition diagram for postcond.

If the tvexpr is either absent or present and true, the command is executed. If the tvexpr is present and false, the command word and its arguments are passed over without execution.

The postcond may also be used to conditionalize the arguments of DO, GOTO, and XECUTE.

---

a108006: format in READ and WRITE

3.5.4 format in READ and WRITE

The format, which can appear in READ and WRITE commands, specifies output format control. The parameters of format are processed one at a time, in left-to-right order.

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

| ? intexpr |

See also the transition diagram for format.

The parameters, which need not be separated by commas when occurring in a single instance of format, may take the following forms.

! 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.)

---

a108007: Side-Effects on $X and $Y

3.5.5 Side Effects on $X and $Y

As READ and WRITE transfer 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

---

a108008: timeout

3.5.6 Timeout

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

timeout ::= : numexpr

See also the transition diagram for timeout.

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 proceeed.

$T 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, $T is set to 1; otherwise, $T 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, $T is set to 1; otherwise, $T is set to 0.

---

a108009: lineref

3.5.7 Line References

The DO and GOTO commands, as well as the $TEXT function, contain in their arguments means for referring to particular lines within any routine (in the case of DO and GOTO) or within the routine executing the line reference (in the case of $TEXT). This section describes the means for making line references.

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.

$$\underline{\text{lineref}} ::= \underline{\text{dlabel}}\ [\ +\ \underline{\text{intexpr}}\ ]$$

See also the transition diagram for lineref.

$$\underline{\text{dlabel}} ::= \left| \begin{array}{c} \underline{\text{label}} \\ @\ \underline{\text{expratom}}\ \underline{\text{V}}\ \underline{\text{dlabel}} \end{array} \right|$$

If + intexpr is absent, the line denoted by lineref is the one containing label in a defining occurrence. If + intexpr is present and has the value n > 0, the line denoted is the nth line after the one containing label in a defining occurrence. A negative value of intexpr is erroneous. When label is an instance of intlit, leading zeros are significant to its spelling.

In the context of DO or GOTO, either of the following conditions is erroneous.

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 and GOTO can refer to a line in a routine other than that in which they occur; this requires a means of specifying a routine name.

$$\underline{\text{routineref}} ::= \left| \begin{array}{c} \underline{\text{routinename}} \\ @\ \underline{\text{expratom}}\ \underline{\text{V}}\ \underline{\text{routineref}} \end{array} \right|$$

The total line specification in DO and GOTO is in the form of entryref.

$$\underline{\text{entryref}} ::= \left| \begin{array}{c} \underline{\text{lineref}}\ [\ \hat{}\ \underline{\text{routineref}}\ ] \\ \hat{}\ \underline{\text{routineref}} \end{array} \right|$$

See also the transition diagram for entryref.

If the delimiter ^ is absent, the routine being executed is implied. If the lineref is absent, the first line is implied.

a108018: Command Definitions

3.6 Command Definitions

The specifications of all commands follow.

B[REAK]
C[LOSE]
D[O]
E[LSE]
F[OR]
G[OTO]
H[ALT]
H[ANG]
I[F]
J[OB]
K[ILL]
L[OCK]
O[PEN]
Q[UIT]
R[EAD]
S[ET]
U[SE]
V[IEW]

W[RITE]
X[ECUTE]
Z[unspecified]

---

a108024: BREAK

3.6.1 BREAK

B[REAK] postcond	[Space] argument syntax unspecified

BREAK provides an access point within the standard for nonstandard programming aids. BREAK without arguments suspends execution until receipt of a signal, not specified here, from a device.

See also the transition diagram for Break.

---

a108025: CLOSE

3.6.2 CLOSE

C[LOSE] postcond Space L closeargument

closeargument ::=	expr [: deviceparameters] @ expratom V L closeargument
deviceparameters ::=	expr (expr [: [expr]] ...)

See also the transition diagram for deviceparameters.

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 empty value.

See also the transition diagram for Close.

---

a108026: DO

3.6.3 DO

D[O] postcond Space doargument

doargument ::=	entryref postcond @ expratom V L doargument

DO is a generalized subroutine call. Each doargument is executed, one at a time in left-to-right order. Execution of a doargument is described below.

a. If postcond is present and false, execution of the doargument is complete at this point. If postcond is absent, or present and true, proceed to the following step.

b. Before proceeding to the next argument of this DO or to the command following this DO, execution continues at the left end of the line specified by the entryref. Execution returns to the argument or command following this argument upon encountering an executed QUIT or eor not within the scope of a subsequently executed doargument or FOR. The scope of this doargument extends to the execution of that QUIT or eor.

See also the transition diagram for Do.

---

a108027: ELSE

3.6.4 ELSE

E[LSE] [$_{\text{Space}}$]

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

See also the transition diagram for Else.

---

a108031: FOR

3.6.5 FOR

F[OR] $_{\text{Space}}$ lvn = forparameter

$$\text{forparameter} ::= \left| \begin{array}{c} \text{expr}_1 \\ \text{numexpr}_1 : \text{numexpr}_2 : \text{numexpr}_3 \\ \text{numexpr}_1 : \text{numexpr}_2 \end{array} \right|$$

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

FOR specifies repeated execution of 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 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$_1$.
   2. Execute 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 scope once; an undefined value for lvn is erroneous.
   7. If lvn > C-B, processing of this forparameter is complete.
   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 scope once; an undefined value for lvn is erroneous.
    7. If lvn < C-B, processing of this forparameter is complete.
    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 scope once; an undefined value for lvn is erroneous.
    5. Set lvn = lvn + B.
    6. Go to 4.

Note that form d. specifies an endless loop. Termination of this loop 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 the outer FOR encompasses all executions of the scope of the inner FOR corresponding to one complete pass through the inner FOR'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 scope not to be executed under their control.

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

Execution of GOTO effects the immediate termination of all FORs in the line containing the GOTO, and it transfers execution control to the point specified.

See also the transition diagram for For.

---

a108032: GOTO

3.6.6 GOTO

G[OTO] postcond $_{\text{Space}}$ L gotoargument

gotoargument ::= doargument

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 present and 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 start of the line it specifies.

See 3.6.5 for a discussion of additional effects of GOTO when executed within the scope of FOR.

See also the transition diagram for Goto.

---

a108033: HALT

3.6.7 HALT

H[ALT] postcond [$_{\text{Space}}$]

First, LOCK with no arguments is executed. Then, execution of this process is terminated.

See also the transition diagram for Halt.

a108034: HANG

3.6.8 HANG

H[ANG] postcond Space L hangargument

```
hangargument ::= |           numexpr           |
                 | @ 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.

See also the transition diagram for Hang.

a108035: IF

3.6.9 IF

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

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

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

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

IF with n arguments is equivalent in execution to n IFs, 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.

See also the transition diagram for If.

a108036: JOB

3.6.9.5 JOB

J[OB] postcond Space L jobargument

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

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

See also the transition diagram for jobparameters.

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

For each jobargument, the JOB command attempts to initiate another MUMPS process. The JOB command initiates this MUMPS process at the left end of the line specified by the entryref. 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 $T is the success of initiating the process. If no timeout is present, the value of $T is not changed, and process execution is suspended until the MUMPS process named in the jobargument is successfully initiated. The meaning of success in either context is defined by the implementation.

See also the transition diagram for Job.

---

a108037: KILL

3.6.10 KILL

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

                      |          glvn                   |
  killargument ::=    |        ( L lvn )                |
                      | @ expratom V L killargument     |
```

The three argument forms of KILL are given the following names.

a. Empty argument list: Kill All.
b. glvn: Selective Kill.
c. (L lvn): Exclusive Kill.

Killing the variable M sets $D(M) = 0 and causes the value of M to be undefined. Any attempt to obtain the value of M while it is undefined is erroneous. Killing a variable whose $D = 0 has no effect except for possible side effects on the naked indicator.

To kill a variable with the unsubscripted name N also kills all subscripted variables with the same name N.

To kill an m-tuply subscripted variable N(v_1, v_2, ..., v_m) with name N and subscript values v_1, v_2, ..., v_m also kills all n-tuply subscripted variables N(v_1, v_2, ..., v_m, v_{m+1}, ..., v_n), for all n > n, with the same N and identical values for the first subscripts. (These derived n-tuply subscripted variables are called the "descendants" of the m-tuply subscripted variable.)

- The Kill All form kills all local variables.
- The Selective Kill form kills the variables named.

In the Exclusive Kill form lvn must not contain subscripts, although lvn may have descendants. Exclusive Kill kills all local variables except those named and their descendants.

If M is not killed but N, a descendant of M, is killed, the killing of N effects the value of $D(M) as follows: if N is not the only descendant of M, $D(M) is unchanged; if M has a defined value $D(M) changes from 11 to 1; if M does not have a defined value $D(M) changes from 10 to 0.

See also the transition diagram for Kill.

---

a108040: LOCK

3.6.11 LOCK

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

                      |  |   nref    |                 |
                      |  | ( L nref )| [ timeout ]     |
  lockargument ::=    |                                |
                      |                                |
                      | @ expratom V L lockargument    |

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

See also the transition diagram for nref.

LOCK provides a generalized interlock facility available to concurrently executing MUMPS 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 to the naked indicator.

Each lockargument specifies a subspace of the total MUMPS name space for which the executing process seeks to make an exclusive claim; the details of this subspace specification are given below. Prior to evaluating and executing each lockargument, LOCK first unconditionally removes any prior claim on any portion of the name space made by the process as the result of a prior execution of LOCK. Then, if a lockargument is present, an attempt is made to claim the entire subspace defined by the lockargument. If this subspace does not intersect the union of all other subspaces claimed at this instant by all other processes defined by the implementor as sharing the interlock facility, the claim is successfully established and execution proceeds to the next lockargument or command. If the subspace defined by the lockargument intersects any other claimed subspace, execution of this process is suspended until all interfering claims are removed by one or more other processes, or, when a timeout is present, until the timeout expires, if that occurs first.

The subspace defined by one lockargument is claimed effectively all at once or not at all; thus, the observance of appropriate conventions on the use of the name space by all concurrently executing processes can eliminate the possibility of races and deadlocks.

If a timeout is present, the condition reported by $T upon resumption of execution is the successful establishment of the claim. If no timeout is present, execution of the lockargument does not change $T.

The subspace of the total name space defined by each lockargument 'is the union of the subspaces defined by each of the name references nref in the lockargument. Each nref specifies its subspace as follows.

a. If the occurrence of 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, ..., s_i) for which N has the same spelling as nref.
b. If the occurrence of nref is subscripted, then the subspace is the set of the following points: one point for the spelling of nref after all subscripts have been evaluated and one point for each descendant of nref. (See KILL for a definition of descendant.)

See also the transition diagram for Lock.

a108043: OPEN

3.6.12 OPEN

O[PEN] postcond Space L openargument

openargument ::= | expr [: openparameters]
 @ expratom V L openargument |

openparameters ::= | deviceparameters [timeout]
 timeout |

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 3.6.2 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 3.6.16)

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 $T is the success of obtaining ownership. If no timeout is present, the value of $T is not changed and process execution is suspended until seizure of ownership has been successfully accomplished.

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.

See also the transition diagram for Open.

a108044: QUIT

3.6.13 QUIT

Q[UIT] postcond [Space]

The end-of-routine mark eor is equivalent to an unconditional QUIT. If the last command of the routine is executed in such a manner as not to transfer control, or if the last command of the routine is an executed DO and control is returned, then the effect of executing off the end of the routine is to execute the QUIT which is implied by the eor.

The effect of executing QUIT in the scope of FOR is fully discussed in 3.6.5

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 doargument or xargument, if not explicitly then implicitly, because the initial activation of a process may be *thought* of as arising from execution of a DO naming the first executed routine of that process. The effect of executing a QUIT in the scope of a doargument or xargument is to return control to the most recently executed doargument or xargument to which control has not yet been returned by a QUIT. What is executed immediately following the QUIT is the command, doargument, or xargument immediately following the doargument or xargument which most recently transferred control and to which control has not yet been returned. Thus, executed doarguments and xarguments are added to a list of pending returns from which execution of a QUIT (not in the scope of FOR) removes entries in last-in, first-out order.

See also the transition diagram for Quit.

a108045: READ

3.6.14 READ

R[EAD] postcond Space L readargument

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

readcount ::= # intexpr
```

The readarguments are executed, one at a time, in left-to-right order.

The top two argument forms cause output operations to the current device; the next two cause input from the current device to the named local variable. If no timeout is present, execution will be suspended until the input message is terminated, either explicitly or implicitly with a readcount. (See 3.6.16 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, $T is affected as follows. If the input message has been terminated at or before the time at which execution resumes, $T is set to 1; otherwise, $T is set to 0.

When the form of the argument is *lvn [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 e: given to lvn is an integer; the mapping between the set of input characters and the set of integer values given to lvn may be defined by the implementor in a device-dependent manner. If timeout is present and the timeout expires, lvn is given the value –1.

When the form of the argument is lvn [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 lvn is the string entered prior to expiration of the timeout; otherwise, the value given to lvn is the entire string.

When the form of the argument is lvn # intexpr [timeout], let n be the value of intexpr. It is erroneous if n < O. 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 nth 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 lvn is the string entered prior to the expiration of the timeout; otherwise, the value given to lvn is the string just described.

When the form of the argument is strlit, that literal is output to the current device, provided that it accepts output.

When the form of the argument is format, the output actions defined in 3.5.4 are executed.

$X and $Y are affected by READ the same as if the command were WRITE with the same argument list (except for timeouts) and with each expr value in each writeagrument equal, in turn, to the final value of the respective lvn resulting from the READ.

See also the transition diagram for Read.

---

a108048: SET

3.6.15 SET

S[ET] postcond $_{\text{Space}}$ L setargument

```
                 |      | setpiece |          |
                 |      |   glvn   | = expr   |
setargument ::=  |      | (L glvn )|          |
                 |                            |
                 | @ expratom V L setargument |
```

setpiece ::= $P[IECE] (glvn , expr$_1$ [, intexpr$_2$ [, intexpr$_3$]])

See also the transition diagram for setpiece.

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 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. The execution of a setargument occurs in the following order.

a. 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. 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.
c. One of the following two 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. For each subscripted glvn of the form
       $N(v_1, v_2, ..., v_n)$
       each variable M whose name is of the form
       $N(v_1, v_2, ..., v_m)$
       for all m < n, as well as the unsubscripted variable N, will be affected as follows.
        a. If M already has a "pointer", that is, if $D(M) has a value of 10 or 11, no change is made to the value of $D(M).
        b. If M has no pointer, that is, if $D(M) has a value of 0 or 1, then it is given a pointer. If $D(M) was 0 it becomes 10, and if $D(M) was 1 it becomes 11.

       The $D value of glvn itself is changed as follows:

0	becomes	1
10	becomes	11
1	remains	1
11	remains	11

That is, the pointer status is not altered, but the variable's value may become defined.

2. If the left-hand side of the set is a setpiece, of the form $P(glvn,d,m,n), the value of expr replaces the mth through the nth pieces of the current value of the glvn, where the value of d is the piece delimiter. Note that both m and n are optional. If neither is present, then m = n = 1; if only m is present, then n = m. If glvn has no current value, the empty string is used as its current value. Note that the current value of glvn is obtained just prior to replacing it. That is, the other arguments of setpiece are evaluated in left-to-right order, and the expr to the right of the = is evaluated prior to obtaining the value of glvn. Let s be the current value of glvn, k be the number of occurrences of d in s, that is, k = max(0,$L(s,d) - 1), and t be the value of expr. The following cases are defined, using the concatenation operator of Sect. 3.3.5.
    a. m > n or n < 1.
       The glvn is not changed and glvn 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,m–1) F(min(m–1,1)) t d $P(s,d,n+1,k+1).

   If the glvn is a global variable, the naked indicator is set at the time that the glvn is given its value. If the glvn is a naked reference, the reference to the naked indicator to determine the name and initial subscript sequence occurs just prior to the time that the glvn is given its value.

See also the transition diagram for Set.

---

a108054: USE

3.6.16 USE

U[SE] postcond Space L useargument

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

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 3.6.2 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.

See also the transition diagram for Use.

---

a108055: VIEW

3.6.17 VIEW

V[IEW] 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.

---

a108056: WRITE

3.6.18 WRITE

W[RITE] postcond Space L writeargument

writeargument ::=
| format |
| 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, the output actions defined in 3.5.4 are executed. Each character of output, in turn, affects $X and $Y as described in 3.5.4 and 3.5.5.

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 the ASCII standard and conventions. Each character of output, in turn, affects $X and $Y as described in 3.5.5.

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.

See also the transition diagram for Write.

---

a108057: XECUTE

3.6.19 XECUTE

X[ECUTE] postcond Space L xargument

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

XECUTE provides a means of executing MUMPS code which arises from the process of expression evaluation.

Each xargument is executed one at a time in left-to-right order. If the postcond in the xargument is false, the xargument has no effect. Otherwise, if the value of expr is x, execution of the xargument is 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
```

See also the transition diagram for Xecute.

---

a108058: Z* commands

3.6.20 Z

Z[unspecified] arguments unspecified

All command words in a given implementation which are not defined in the standard are to begin with the letter Z. This convention protects the standard for future enhancement.

---

a10t001: Transition Diagrams

MUMPS LANGUAGE STANDARD
Part II: MUMPS Transition Diagrams

Introduction to MUMPS Transition Diagrams

This document presents the MUMPS dynamic syntax in transition diagram form. This type of representation was introduced (Note 1) for computer programming language definition by Melvin E. Conway and was applied to MUMPS in an early specification (Note 2) .

The diagrams serve as a comprehensive implementation outline for the MUMPS language (Note 3). It is possible to implement the MUMPS language directly from these diagrams, although specific implementation techniques are not stipulated in semantic actions of the diagrams. The necessary operations are given here, but their detailed implementation is left to individual implementors.

One departure from the MUMPS Language Specification is made in the transition diagrams. Syntax checking is performed after false post-conditionals and after the IF and ELSE commands. This was done in order to simply the presentation of the diagrams.

---

a10t002: Scanning Algorithm (text)

1.1 Scanning Algorithm

A transition diagram is a network of nodes and directed paths with at least one entrance node (indicated by the symbol Circle E or the symbol Triangle N, where n is an integer such as 1, 2, 3, etc.) and at least one exit node indicated by one of the following four symbols Circle W, Circle X, Circle Y, Circle Z; Circle X is the "normal" exit).

Each transition diagram defines a syntactic type, whose name is written in underscored lower-case letters, e.g., expr. A string being scanned through a "window" which looks at one character at a time is declared to be an instance of the syntactic type defined by a diagram if and only if the algorithm given below exactly scans the string while traversing the transition diagram from an entrance node to an exit node.

Each directed path from one node to another node is either associated with a symbol "on" that path, or the path is "blank". If a path has a symbol on it, the symbol can be either the name of a syntactic type defined by a transition diagram, implying a "call" (possibly recursive) to that diagram, or a symbol from the primitive alphabet. The primitive alphabet consists of the 95 ASCII graphics, including SP (space), plus ls, eol, eor, and eoi (the eoi character is used to indicate the end of argument or sub-argument level indirection; its actual form is not defined).

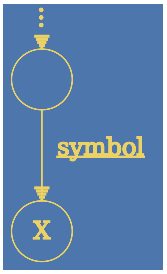
The scanning algorithm works as follows. In the next paragraph, the rule for following a path from one node to the next node is given. Once that can be done, the rule is applied iteratively starting at an entrance node until an exit node is encountered. At this point, a call to the diagram just traversed has been completed, and the path which made the call may be traversed. If a dead end is reached, and the window has not moved since entering the diagram, it only means that the call to this diagram has not succeeded and another path must be attempted. If a dead end is reached after moving the window over at least one input symbol since the entrance node, this is the indication of a syntax error.

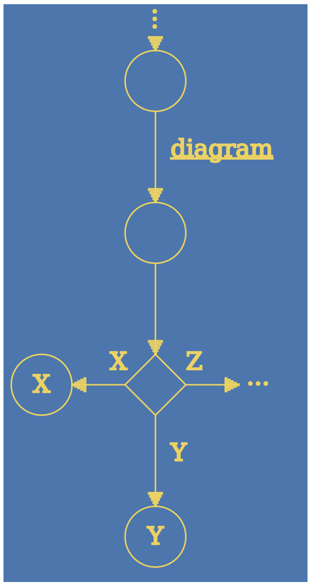
The rule for leaving a node is as follows. All nonblank paths are tried, with primitive symbols tried first, then calls to other diagrams. The blank path, if it is present, provides the default case and is taken only after all other paths fail. A path with a primitive symbol on it may be traversed if and only if the symbol in the window equals the symbol on the path. In this case, the path is traversed and the window is moved one position to the right. A path with a call to a transition diagram may be traversed if and only if a call to that diagram results in successfully reaching an exit node of that diagram. Any window movement arising from a call to a diagram will be done within the called diagram.

Any path can direct the performance of an action, indicated by a number in a square box on the path. The action may be performed after the path is traversed. Certain actions, called "privileged actions", are always executed after traversing the path on which they appear; all other actions are executed only if both the semantic action flags Linact and Comact are True (see the scanning algorithm). Note that the actions within a called diagram precede the action specified on the path of the call.

If a diagram contains only one exit type, the X type is used, If it contains W, X, Y and/or Z, the exit actually used as a result of a given call may be tested by the caller. The notation used is as follows.

single-exit diagram multiple-exit diagram





Actions generally make reference to temporary variables, such as A, B, C, D (see the diagram for the $PIECE function). These variables are strictly local to each invocation of a diagram. A communication variable "Result" is used to pass values among diagrams.

Two branching notations are used in the diagrams. These are illustrated below.

~~svg~~/image/AnnoStd/1984_p03a.svg~~130~~150~~ ~~svg~~/image/AnnoStd/1984_p03b.svg~~130~~150~~

p03a p03b
here there

Notation 1 indicates that this syntactic construct continues at the Oentrance of the named diagram. In other words, whenever the O (branch) symbol is encountered, the logical flow is transferred to the beginning of the diagram whose name appears to the right of the O . This construct is merely a convenient notation for presentation of the diagrams. Notation 2 indicates a transfer of control to a specific entry oint (other than the OE entrance) in the named diagram. The n in the symbol is an integer which indicates the number of the entry point in the diagram whose name appears to the right of the {n} This construct is used primarily to show clearly the control flow in the M MPS language from DO, FOR, and XECUTE commands.

One other construct used in the diagrams is show below

~~svg~~/image/AnnoStd/1984_p03c.svg~~170~~250~~

p03c
here

The logical value of "condition" is tested. If it is True, path 1 is taken. If it is False, path 2 is taken.

The scanning algorithm on the next page is used to move from one "syntax node" to the next (a syntax node is denoted by a circle). Any of the intervening branching or testing nodes discussed above are taken automatically. The variable "case" in the algorithm is used for determining the actual exit from a diagram call with multiple exit. Underlined parts of the algorithm represent operations which have not undergone detailed stepwise refinement.

To execute a MUMPS routine, the window is initially positioned to the first character of the text of the routine, anndthe routine diagram in Section 2 is invoked. The algorithm below then scans through the routine text using this diagram and all subsequently called diagrams.

a10t003: Scanning Algortitm (Pascal-like)

Transition Diagram Scanning Algorithm (Pascal-like)

```
TYPE alpha - ARRAY[1..16) OF char;
VAR linact, comact : Boolean; case : char;
FUNCTION traversediagram(diagramname : alpha) : integer;
TYPE exitnode - (X,Y,Z,W);
VAR oneprimitivetraversed, continuescanning, pathtaken, pathexists : Boolean;
    exittype : integer;
BEGIN oneprimitivetraversed FALSE; continuescanning TRUE;
|   WHILE continuescanning DO
|       BEGIN
|       |   pathtaken := FALSE;
|       |   pathexists TRUE;
|       |   WHILE pathexists DO
|       |   |   BEGIN
|       |   |   |   find next path from this syntax node with a primitive symbol on it;
|       |   |   |   IF such a path found THEN
|       |   |   |   |   BEGIN
|       |   |   |   |   |   IF character on the path - character in the window THEN
|       |   |   |   |   |       BEGIN
|       |   |   |   |   |       |   exittype := 1;
|       |   |   |   |   |       |   oneprimitivetraversed := TRUE;
|       |   |   |   |   |       |   mark this path as the chosen path to be taken;
|       |   |   |   |   |       |   move window one position to the right;
|       |   |   |   |   |       |   IF input string is exhausted THEN syntax error
|       |   |   |   |   |       |   ELSE
|       |   |   |   |   |       |       BEGIN
|       |   |   |   |   |       |       |   pathtaken TRUE;
|       |   |   |   |   |       |       |   pathexists FALSE;
|       |   |   |   |   |       |       END
|       |   |   |   |   |       END
|       |   |   |   |   END
|       |   |   |   ELSE pathexists :- FALSE;
|       |   |   END;
|       |   IF ¬ pathtaken THEN
|       |   |   BEGIN
|       |   |   |   pathexists := TRUE;
|       |   |   |   WHILE pathexists DO
|       |   |   |       BEGIN
|       |   |   |       |   find next path from this syntax node with a call to "diagramname";
|       |   |   |       |   IF such a path found THEN
|       |   |   |       |   |   BEGIN
|       |   |   |       |   |   |   exittype := traversediagram("diagramname");
|       |   |   |       |   |   |   IF exittype THEN
|       |   |   |       |   |   |       BEGIN
|       |   |   |       |   |   |       |   pathtaken := TRUE;
|       |   |   |       |   |   |       |   pathexists FALSE;
|       |   |   |       |   |   |       |   mark this path as the chosen path to be taken;
|       |   |   |       |   |   |       END
|       |   |   |       |   |   END
|       |   |   |       |   ELSE pathexists :- FALSE;
|       |   |   |       END
|       |   |   END;
|       |   IF ¬(pathtaken) ∧ there is a blank path from this syntax node THEN
|       |   |   BEGIN
|       |   |   |   exittype := 1;
|       |   |   |   pathtaken := TRUE;
|       |   |   |   nark this path as the chosen path to be taken;
|       |   |   END;
|       |   IF pathtaken THEN
|       |   |   BEGIN
|       |   |   |   CASE exittype OF
|       |   |   |   |   1: case := 'X';
|       |   |   |   |   2: case := 'Y';
|       |   |   |   |   3: case := 'Z';
|       |   |   |   |   4: case := 'W';
|       |   |   |   END;
|       |   |   |   find marked path from this syntax node and move to next syntax node along it;
|       |   |   |   IF path traversed specified an acti:a THEN
|       |   |   |   |   BEGIN
|       |   |   |   |   |   IF (linact ∧ comact) ∨ action is a privileged action THEN DO action;
|       |   |   |   |   END;
|       |   |   |   IF next syntax node is an exitnode THEN
|       |   |   |       BEGIN
|       |   |   |       |   continuescanning := FALSE;
|       |   |   |       |   CASE exitnode OF
|       |   |   |       |   |   X: traversediagram := 1;
|       |   |   |       |   |   Y: traversediagram := 2;
|       |   |   |       |   |   Z: traversediagram := 3;
|       |   |   |       |   |   W: traversediagram := 4;
|       |   |   |       |   END
|       |   |   |       END
|       |   |   END
|       |   ELSE
|       |       BEGIN
|       |           IF oneprimitivetraversed THEN syntax error
|       |           ELSE
|       |               BEGIN
|       |               |   traversediagram := 0;
|       |               |   continuescanning := FALSE;
|       |               END
|       END;
END;
```

a10t004: Example

1.2 Example

A simple arithmetic expression, sum, can be defined as follows.

sum P[+ sum]

The primitive symbols are P and +,

This example can be used to illustrate recursion and the technique for scanning text with a transition diagram. The definition above forces a right-to-left order of evaluation. Because this definition is recursive, a "Stack" is needed to save intermediate results whenever sum reinvokes itself. Items are saved onto this Stack by a PUT operation, and are retrieved by a GET operation, in a last-in-first-out order. The transition diagram for the definition of sum above is shown below.

~~svg~~/image/AnnoStd/1984_p05.svg~~250~~250~~

p05 diagram
here

1. Place value of P into A, that is, P Right Arrow A.
2. A + Result Right Arrow Result.
3. A Right Arrow Result.

In order to illustrate the effect of a diagram's structure on the order of evaulation, this example can be tested on the input string P+Q+R eol , where Q and R are separate occurrences of the symbol P, and eol is the string termination character. The steps below interpret this string using the above diagram, in the same fashion as the algorithm on the previous page uses the MUMPS diagrams in Section 2.

1. The window is initially positioned at the first character of the string, P.
2. Path Ea contains the same symbol as in the window (P); consequently, the path to node a is traversed and the window is moved to the next character of the input string.

3. Action 1 is executed. The value of P is placed in A_0. (A_0 is the zero-level occurrence of temporary variable A.)
4. Path ab contains the same symbol as in the window (+), causing transversal of the path to node b, and moving the window to the next character of the input string.
5. Path bX is a call to sum. PUT Ao on the Stack and start at node E, now at level 1.
6. Path Ea contains the same symbol as in the window (Q=P), so traverse the path to node a, moving the window to the next input character.
7. Action 1 is executed, placing the value of Q in A_1.
8. Path ab contains the same symbol as in the window (+), so traverse the path to node b, and move the window to the next character.
9. Path bX is again a call to sum. PUT A_l on the Stack and start at node E, now at level 2.
10. As before, path Ea is traversed and the value of R is placed in A_2.
11. The default blank path aX is now traversed because there is an end-of-line symbol in the window, which is not the same as a +.
12. Action 3 is executed, which places the value in A_2 (i.e., R) into Result.
13. The second call to sum is now complete, so effectively path bX can now be traversed at level 1.
14. Action 2 is executed, by first performing a GET A_l from the Stack, then forming A_l + Result (which is Q + R), and finally placing the sum into Result.
15. The first call to sum is now successful, so path bX can now be traversed at level 0.
16. As above, action 2 is executed, doing a GET Ao from the Stack, forming the sum Ao + Result (which is P + Q + R), and placing this value into Result.
17. The variable Result now contains the value obtained from scanning the input string P+Q+R eol. This value can in turn be used in another diagram which invoked the sum diagram, much as Result was used in the recursive calls of sum above.

---

a10t005: Common Data

1.3 Common Data Used by the MUMPS Diagrams

The following variables are common to the actions used by all of the MUMPS diagrams of Section 2.

1. **naked indicator**. This is the n-tuple described in Section 3.2.3 of MDC/28. It is initially undefined and becomes undefined under certain circumstances discussed in Section 3.2.3.
2. **Devicelist**. This is an n-tuple of device names with associated device parameters. It is added to whenever a new device is successfully secured with the OPEN command. It is updated when the device parameters of a previously owned device are changed with the OPEN, CLOSE, or USE commands.
3. **Openlist**. This is an n-tuple of device names which are created by the OPEN command and can be removed with the CLOSE command.
4. **Devnam**. This is the device name from an argument of the OPEN, CLOSE, or USE command. It is used by deviceparameters to get default values for omitted parameters.
5. **Argind**. This is the argument-level indirection flag, It is initially False, and is set True if argument-level indirection is encountered. It is saved each time argument-level indirection is detected.
6. **Nameind**. This is the name-atom (sub-argument) level indirection flag. It is initially False, and is set True if name-atom indirection is encountered. It is saved each time name-atom indirection is detected.
7. **Indsw**. This is the command-level indirection counter. It is initially zero, is incremented each time command-level indirection is detected, and is decremented when an eor is encountered or a QUIT is executed under command-level indirection.
8. **Forsw**. This is the FOR command counter. It is initially zero, is incremented each time a FOR command is encountered, and decremented when a FOR has exhausted its FOR list. It is also decremented appropriately by the QUIT and GOTO commands.
9. **Dosw**. This is the DO command counter. It is initially zero, is incremented each time a DO command is encountered, and decremented when a DO is completed, either from a QUIT or an eor.
10. **Ifswitch**. This is the name used in the diagrams for the special variable $TEST. It is initially 0 (False).
11. **Linact**. This has the value True or False. If False, all semantic actions are inhibited for all commands until an eol is detected (see the diagrams for the IF and ELSE commands). Linact is set True prior to executing a new line of a routine.
12. **Comact**. This has the value True or False. If False, all semantic actions are inhibited for the duration of the command (see the diagram for postcond). Comact is set True prior to executing a new command.
13. **Present routine name**. This defines the scope of label values. It is saved when a DO or XECUTE command is executed for the return.
14. **Window position**. This is a pointer which gives the current character position being scanned. It is saved whenever a DO or FOR command is executed, and when any indirection is encountered.
15. **Result**. This is used to pass values among various diagrams.
16. **Timeout**. This is used to set up a time limit for the following: (1) timed-length reads in the READ command,

(2) securing device ownership in the OPEN command, and (3) interlocking software resources in the LOCK command. It is initially 0.

17. **Setsw**. This is a flag which is always False unless a SET command is being executed. It is used in the glvn diagram to determine whether or not the naked indicator is immediately affected by a global reference.
18. **Indcom**. This is a communications flag used only by indarg and indnam. It is always False unless name level indirection is detected in indarg.

Additionally, a **Stack** is used in the diagrams to explicitly show the mechanism by which indirection is handled and the transfer of control commands are executed. It is a simple push-down stack operating on a last-pushed, first-pulled basis. Items pushed onto the Stack are listed in the semantic actions following a PUT directive. Items pulled off the Stack are listed in the semantic actions following a GET directive. A RESET directive is used in a few semantic actions to indicate items whose values are recovered from the Stack without actually removing them from the Stack.

---

a10t006: Diagrams

2. MUMPS Transition Diagrams

The transition diagrams on the following pages are organized in a "top-down" manner. The first diagram invoked is always the routine diagram; it is therefore the highest level diagram. The routine diagram then invokes the line diagram, which may in turn invoke the command diagram, and so on.

Thus, the logical flow is reflected in the order of presentation of the diagrams. For ease of reference, the individual command and function diagrams are organized alphabetically. A number of command primitive diagrams appear after the command diagrams. These are also organized in a top-down logical manner, as much as possible. The expression diagrams and expression atom diagrams are similarly arranged.

---

a10t007: routine

routine

See routine for formal definition.

~~svg~~/image/AnnoStd/1984_trans_routine.svg~~250~~445~~

---

routine diagram

here

a10t008: line

line

See line for formal definition.

~~svg~~/image/AnnoStd/1984_trans_line.svg~~600~~900~~

line diagram

here

1. Set Linact = True.
2. Set Forsw = 0,
   Comact = True,
   Argind = False.
3. (Privileged) Set Comact = True.
4. (Privileged) Set Linact = True.
5. RESET FOR lvn name, FOR body position, loop counters (if any), and FOR window position from Stack.
   The value of entry i going to the FOR command is the number of loop counters + 1.
   Link window to FOR window position.

---

a10t009: command

command

See command for formal definition.

~~svg~~/image/AnnoStd/1984_trans_command.svg~~1050~~2100~~

command diagram
here

Note: In the command diagram, the corresponding lower-case letter is considered equivalent to the upper-case letter shown for each command name.

---

a10t010: BREAK

BREAK command

See Break for formal definition.

~~svg~~/image/AnnoStd/1984_trans_break.svg~~300~~850~~

break diagram
here

1. Suspend operation until receipt of the proceed-from-break signal.

---

a10t011: CLOSE

CLOSE command

See Close for formal definition.

~~svg~~/image/AnnoStd/1984_trans_close.svg~~420~~750~~

CLOSE diagram
here

1. Result Right Arrow Devnam.
2. Null string ("") Right Arrow Result.
3. Search the Openlist for the device named in Devnam.
   If not found, take no further action.
   Otherwise, perform the following operations:
   a. Remove the device specified in Devnam from the Openlist.
   b. If Result contains any device parameters, find the device named in Devnam in the Devicelist, and change those parameters which appear in Result to their new values from Result.
   c. Perform any termination procedures for the device named in Devnam according to its device parameters in Devicelist.
   d. If the named device is the current device ($IO), execute an OPEN P USE P where P designates a predetermined default device.

---

a10t012: DO

DO command

See Do for formal definition.

~~svg~~/image/AnnoStd/1984_trans_do.svg~~450~~750~~

DO diagram
here

1. Result Right Arrow A.
2. Dosw + 1 Right Arrow Dosw.
   PUT Present routine name, Window position, Forsw, Indsw, and Argind on Stack.
   Load routine named in A if necessary.
   Set Indsw = 0.
   Position window to the first character of the entry reference named in A.
3. Dosw - 1 Right Arrow Dosw.
   GET Argind, Indsw, Forsw, Window position, and Present routine name from Stack.
   Load routine if necessary.
   Link window to retrieved Window position.

---

a10t013: ELSE

ELSE command

See Else for formal definition.

~~svg~~/image/AnnoStd/1984_trans_else.svg~~165~~550~~

ELSE diagram
here

1. Set semantic action flag Linact False.

Note: If Ifswitch ($TEST) is True (1), then Linact will be set False, which will inhibit execution of all semantic actions for all commands until a new line is encountered. However, all syntactic paths for all commands are traversed (that is, syntax checking is performed).

---

a10t014: FOR

FOR command

See For for formal definition.

~~svg~~/image/AnnoStd/1984_trans_for.svg~~1010~~965~~

FOR diagram
here

1. Forsw + 1 Right Arrow Forsw.
   Result Right Arrow B.
   PUT Result (FOR lvn name) eol onto Stack.
   PUT Window position + 1 on Stack.
2. Window position (FOR body position) Right Arrow A.
   GET Window position from Stack.
   PUT A on Stack.
3. Result Right Arrow A.
4. Result Right Arrow B.
   PUT B on Stack.
   Take the numeric interpretation of the value in A.
   That is, apply the rules given in Section 3.2.5 of Part I to A.
5. Result Right Arrow C.
   PUT C on Stack.
6. Place the result in A in the FOR lvn name.
7. Current Window position Right Arrow D.
   RESET FOR body position from Stack and link window to it.
   PUT D on Stack.
8. Place value of retrieved FOR lvn name into A, value of 1st retrieved loop counter into B.
   A + B Right Arrow A.
   Place the result in A in the FOR lvn name.

Link window to FOR body position.

9. A + B Right Arrow A.
   Place the result in A in the FOR lvn name.
   Link window to FOR body position.
10. Place value of retrieved FOR lvn name into A, value of 1st retrieved loop counter into B, value of 2nd retrieved loop counter into C.
11. GET FOR window position and loop counters (if any) from Stack.
12. Do action 11.
    GET FOR body position and FOR lvn name from Stack.
    Forsw - 1 Right Arrow Forsw.
    Set semantic action flag Linact False.

---

a10t015: GOTO

GOTO command

See Goto for formal definition.

~~svg~~/image/AnnoStd/1984_trans_goto.svg~~700~~850~~

GOTO diagram
here

1. Result Right Arrow A.
2. GET previous level's Argind and Window position from the Stack.
   GET the indirect argument off the Stack.
3. GET the FOR information off the Stack.
   Forsw - 1 Right Arrow Forsw.
4. Load routine named in A if necessary.
   Then position window to the first character of the entry reference named in A.

---

a10t016: H* commands

H commands

~~svg~~/image/AnnoStd/1984_trans_halthang.svg~~520~~375~~

HALT or HANG diagram
here

a10t017: HALT

HALT command

See Halt for formal definition.

~~svg~~/image/AnnoStd/1984_trans_halt.svg~~350~~450~~

HALT diagram
here

1. Perform a LOCK with no arguments (action 1 of the LOCK command).
   Then terminate execution.

---

a10t018: HANG

HANG command

See Hang for formal definition.

~~svg~~/image/AnnoStd/1984_trans_hang.svg~~250~~800~~

HANG diagram
here

1. If Result > 0, suspend execution for the number of seconds specified by the value in Result.

---

a10t019: IF

IF command

See If for formal definition.

~~svg~~/image/AnnoStd/1984_trans_if.svg~~450~~850~~

IF diagram
here

1. If Result = 0 then False Right Arrow Ifswitch;
   otherwise True Right Arrow Ifswitch ($TEST).
2. Set semantic action flag Linact False.

Note: If Ifswitch ($TEST) is False (0), then Linact will be set False; which will inhibit execution of all semantic actions for all commands until a new line is encountered. However, all syntactic paths for all commands are traversed (that is, syntax checking is performed).

---

a10t020: JOB

JOB command

See Job for formal definition.

~~svg~~/image/AnnoStd/1984_trans_job.svg~~575~~750~~

JOB diagram
here

1. Result Right Arrow A,
   Null string ("") Right Arrow B.
2. Result Right Arrow B.
3. Attempt to initiate another MUMPS process. If B contains any job parameters, use them as specified by the implementation to initialize characteristics of the new MUMPS process. Initiate a distinct MUMPS process from this one, load the routine named in A (or the current routine if A does not contain a routine name) into that process, and position Window to the first character of the entry reference named in A. This operation suspends execution until it succeeds.
4. Set up a timer of Timeout seconds. Attempt to initiate another MUMPS process, as described in action 3, at least once, and then repeatedly until it succeeds, or until the timer expires, whichever occurs first. (The meaning of success is defined by the implementation.)
5. False Right Arrow Ifswitch ($TEST).
6. True Right Arrow Ifswitch ($TEST).

---

a10t021: KILL

KILL command

See Kill for formal definition.

~~svg~~/image/AnnoStd/1984_trans_kill.svg~~500~~750~~

KILL diagram
here

1. Kill all local variables.
2. Kill the variables whose names contain the name in Result.
3. Check to see that Result is at most a 1-tuple (that is, that the local variable is not subscripted).
   If it is not, trap execution.
   Otherwise, mark the local variables whose names contain the name in Result.
4. Kill all local variables except those marked by action 3.
   Remove the marks.

Note: The n-tuple name (a_1, a_2, ..., a_n) contains the m-tuple name (b_1, b_2, ..., b_m) if and only if $m < n$, and for each i where i = 1, 2, ..., m, $a_i = b_i$.

---

a10t022: LOCK

LOCK command

See Lock for formal definition.

~~svg~~/image/AnnoStd/1984_trans_lock.svg~~550~~850~~

LOCK diagram
here

1. Remove all claims on the MUMPS name spae from prior LOCKs.
2. Null string ("") Right Arrow A.
3. Replace the n-tuple in A by the n + 1-tuple (A:Result).
4. Result Right Arrow A.
5. Do action 1.
   Then attempt to claim the subspace of all names in A.
   This action suspends execution until it succeeds.
6. Set up a timer of Timeout seconds.
   Do action 1.
   Then attempt to claim the subspace of all names in A at least once, and then repeatedly until the claim succeeds or the timer expires, whichever occurs first.
7. False Right Arrow Ifswitch ($TEST).
8. True Right Arrow Ifswitch ($TEST).

---

a10t023: OPEN

OPEN command

See Open for formal definition.

~~svg~~/image/AnnoStd/1984_trans_open.svg~~575~~750~~

OPEN diagram
here

1. Result Right Arrow Devnam,
   Null string ("") Right Arrow B.
2. Result Right Arrow B.
3. Search the Openlist for the device named in Devnam.
   If found, perform only operation c. below and take no further action.
   Otherwise, perform all the following operations:
   a. Attempt to seize exclusive ownership of the device named in Devnam.
      This operation suspends execution until it succeeds.
   b. Add the specified device to the Openlist.
   c. If B contains any device parameters, find the device named in Devnam in the Devicelist, and change

    those parameters which appear in Result.
    d. Perform any initiation procedures for the device named in Devnam according to its device parameters in Devicelist.
4. Set up a timer of Timeout seconds.
   Search the Openlist for the device named in B.
   If found, perform operation 3c and indicate that the timer has not expired.
   Otherwise, attempt to seize exclusive ownership of the device named in Devnam at least once, and then repeatedly until it succeeds, or the timer expires, whichever occurs first.
   If ownership is established prior to expiration of the timer, perform actions 3b, 3c and 3d.
5. False Right Arrow Ifswitch ($TEST).
6. True Right Arrow Ifswitch ($TEST).

---

a10t024: QUIT

QUIT command

See Quit for formal definition.

~~svg~~/image/AnnoStd/1984_trans_quit.svg~~650~~785~~

QUIT diagram
here

1. GET current FOR information off the Stack.
   Forsw - 1 Right Arrow Forsw.
   Set semantic action flag Linact False.
2. RESET FOR lvn name, FOR body position, loop counters (if any), and FOR window position from Stack.
   The value of entry i going to the FOR command is the number of loop counters + 1.
   Link window to FOR window position.
3. Terminate execution.

---

a10t025: READ

READ command

See Read for formal definition.

~~svg~~/image/AnnoStd/1984_trans_read.svg~~510~~750~~

Read diagram
here

1. Result Right Arrow A.
2. Zero ("0") Right Arrow B.
3. If Result < 1, trap execution.
   Otherwise, Result Right Arrow B.
4. Wait for input from the current device ($IO).
   Proceed either after receiving B characters, if B > 0, or after receipt of EOM (End-of-Message), and place input string into variable named in A.
   $X + $L(A) Right Arrow $X.
5. Wait for input from the current device.
   Proceed after receipt of one character and place the integer character code into variable in A.
   $X + 1 Right Arrow $X.
6. Set up a timer of Timeout seconds.
   Check for receipt of EOM, or receipt of B characters, if B > 0, and proceed if received.
   Otherwise, proceed after receipt of EOM, receipt of B characters, if B > 0, or expiration of timer, whichever occurs first.
7. Set up a timer of Timeout seconds.
   Check for receipt of one character and proceed if received.
   Otherwise, proceed after receipt of one character or expiration of timer, whichever occurs first.

8. Place input string into variable named in A.
   True Right Arrow Ifswitch ($TEST).
   $X + $L(A) Right Arrow $X.
9. Place the integer character code of input into variable named in A.
   True Right Arrow Ifswitch ($TEST).
   $X + 1 Right Arrow $X.
10. Place the input string so far received into variable named in A.
    False Right Arrow Ifswitch ($TEST).
    $X + $L(A) Right Arrow $X.
11. Place –1 into variable named in A.
    False Right Arrow Ifswitch ($TEST).
12. Output Result to the current device.
    $X + $L(Result) Right Arrow $X.

---

a10t026: SET

SET command

See Set for formal definition.

~~svg~~/image/AnnoStd/1984_trans_set.svg~~500~~850~~

SET diagram
here

1. Set Setsw = True.
2. Null string ("") Right Arrow B.
3. Replace the n-tuple in B with the n + 1-tuple (B:Result).
4. Result Right Arrow B.
5. For each of the variables named in B, proceeding from left-to-right, perform the following operations:
   a. Place the next variable in B into A.
   b. If the variable in A is a global variable, do action 3 of glvn.
   c. Place the value in Result in the variable named in A. If the variable named in A has the form of a setpiece, the details of this action are described in Section 3.6.15 of Part I.

---

a10t027: USE

USE command

See Use for formal definition.

~~svg~~/image/AnnoStd/1984_trans_use.svg~~400~~750~~

USE diagram
here

1. Result Right Arrow Devnam.
2. Null string ("") Right Arrow Result.
3. Search the Openlist for the device named in Devnam.
   If not found, trap execution. Otherwise, perform the following operations:
   a. Set $IO to the device named in Devnam, and make this device the current device for all input and output.
   b. If Result contains any device parameters, find the device named in Devnam in the Devicelist, and change those parameters which appear in Result to their new value from Result.
   c. Perform any initialization procedures for the device named in Devnam according to its device parameters in Devicelist.

---

a10t028: WRITE

WRITE command

See Write for formal definition.

~~svg~~/image/AnnoStd/1984_trans_write.svg~~375~~650~~

WRITE diagram
here

1. Output to the current device the value of Result as a string.
   The effect of this string at the device is defined by the ASCII Standard and conventions.
   Update $X and $Y as described in Section 3.5.5 of MDC/28.
2. Output the character whose character code is in Result.
   This output may be performed in a device-dependent manner.

---

a10t029: XECUTE

XECUTE command

See Xecute for formal definition.

~~svg~~/image/AnnoStd/1984_trans_xecute.svg~~450~~750~~

XECUTE diagram
here

1. Concatenate (ls, Result, eol) Right Arrow A.
   Concatenate (A, ls, "QUIT", eol) Right Arrow A.
2. Indsw + 1 Right Arrow Indsw.
   PUT A on Stack.
   PUT Present routine name, Window position, Forsw, and Argind on Stack.
   Position window to the first character of the indirect string on Stack.
3. Indsw - 1 Right Arrow Indsw.
   GET Argind, Forsw, Window position, and Present routine name from Stack.
   GET the indirect string off the Stack.
   Load routine if necessary.
   Link window to retrieved Window position.

---

a10t030: postcond

postcond

See postcond for formal definition.

~~svg~~/image/AnnoStd/1984_trans_postcond.svg~~150~~450~~

postcond diagram
here

1. Set semantic action flag Comact False.

Note: If argcond returns a Circle Y condition (tvexpr is False), postcond inhibits execution of all semantic actions for the command until the arguments terminate. However, all syntactic paths are traversed as normal (that is, syntax checking is performed).

---

a10t031: argcond

argcond

~~svg~~/image/AnnoStd/1984_trans_argcond.svg~~250~~470~~

1. Result Right Arrow A.

Note: The argcond diagram scans off the optional post-conditional wherever it may appear, both after command names and within command arguments. The meanings of the exits from argcond are:
X condition is true; execute argument
Y condition if false; skip argument.

---

a10t032: argument

argument

See argument for formal definition.

~~svg~~/image/AnnoStd/1984_trans_argument.svg~~250~~450~~

Note: The argument diagram is used to scan off the delimiter between the command word and its arguments (if any). It also handles indirection on the first argument of a command.

---

a10t033: comend

comend

~~svg~~/image/AnnoStd/1984_trans_comend.svg~~500~~550~~

1. (Privileged) GET previous level's Argind and Window position from the Stack.
   GET the indirect argument from the Stack.
   Link window to retrieved Window position.

Note: The comend diagram is used to scan off the delimiter after each argument of a command. It also handles termination of argument level indirection, and tests for argument-level indirection in the next argument (if one is present).

---

a10t034: indarg

indarg

~~svg~~/image/AnnoStd/1984_trans_indarg.svg~~380~~650~~

1. (Privileged) False Right Arrow A. If character in window is , (comma), SP (space), eol, or eoi, then True Right Arrow A.
2. If Result contains an SP (space) not within quotes, or an eol or eoi, trap execution. Otherwise, append an eoi to Result, then PUT Result and current Window position on the Stack. Position window to the first character of the indirect argument.
3. PUT Argind on Stack.
   True Right Arrow Argind.
4. If character in window is not an alpha, digit, % (percent), @ (commercial at), or ^ (circumflex), trap execution.
   Otherwise, PUT Nameind on Stack.
   True Right Arrow Nameind.
5. (Privileged) True Right Arrow Indcom (used to indicate that name level indirection has been detected and scanned while syntax checking; see the indnam diagram).

---

a10t035: indnam

indnam

~~svg~~/image/AnnoStd/1984_trans_indnam.svg~~280~~650~~

1. PUT Nameind on the Stack.

False Right Arrow Nameind.

2. GET Nameind off the Stack.
3. Append an eoi to Result, then PUT Result and current Window position on the Stack.
   Position window to the first character of the indirect name.
   PUT Nameind on the Stack.
   True Right Arrow Nameind.
4. (Privileged) False Right Arrow Indcom.

---

a10t036: format

format

See format for formal definition.

~~svg~~/image/AnnoStd/1984_trans_format.svg~~450~~450~~

1. Output top-of-form operation on current device.
   Place 0 Right Arrow $X,
   0 Right Arrow $Y.
2. Output new line operation on current device.
   Place 0 Right Arrow $X,
   $Y + 1 Right Arrow $Y.
3. Result Right Arrow A.
   Tab to column A on current device; that is, output max(0, A - $X) spaces.
   Place max($X, A) Right Arrow $X.

---

a10t037: timeout

timeout

See timeout for formal definition.

~~svg~~/image/AnnoStd/1984_trans_timeout.svg~~120~~250~~

1. If Result > 0, Result Right Arrow Timeout.
   Otherwise, 0 Right Arrow Timeout.

---

a10t038: entryref

entryref

See entryref for formal definition.

~~svg~~/image/AnnoStd/1984_trans_entryref.svg~~275~~850~~

1. Null string ("") Right Arrow A.
2. Result Right Arrow A.
3. Place the 2-tuple (A,"") Right Arrow Result.
4. GET previous level's Namind and Window position from the Stack.
   GET the indirect name off the Stack. Link window to retrieved Window position.
5. Place the 2-tuple (A,Result) Right Arrow Result.

Note: An entryref is an ordered pair of the form (a,b), where a is a label + offset and b is a routine name. If a is null, the interpretation of (a,b) is the first line of routine b. If b is null, the interpretation is label a in the present routine. If neither is null, the interpretation is label a of routine b.

---

a10t039: lineref

lineref

See lineref for formal definition.

~~svg~~/image/AnnoStd/1984_trans_lineref.svg~~250~~950~~

1. GET previous level's Nameind and Window position from the Stack.
   GET the indirect name off the Stack.
   Link window to retrieved Window position.
2. Result Right Arrow A.
3. Place the 2-tuple (A + 0) Right Arrow Result.
4. If Result < 0, trap execution.
   Otherwise, place the 2-tuple (A + Result) Right Arrow Result.

Note: A lineref is of the form label + offset, where offset is a positive integer n denoting the nt line after the one containing label.

---

a10t040: label

label

See label for formal definition.

~~svg~~/image/AnnoStd/1984_trans_label.svg~~250~~250~~

---

a10t041: name

name

See name for formal definition.

~~svg~~/image/AnnoStd/1984_trans_name.svg~~255~~350~~

1. Percent ("%") Right Arrow A.
2. Char Right Arrow A.
3. Concatenate (A, Char) Right Arrow A.
4. A Right Arrow Result.

---

a10t042: intlit

intlit

See intlit for formal definition.

~~svg~~/image/AnnoStd/1984_trans_intlit.svg~~170~~250~~

1. Char Right Arrow A.
2. Concatenate (A, Char) Right Arrow A.
3. A Right Arrow Result.

---

a10t043: dummylist

dummylist

~~svg~~/image/AnnoStd/1984_trans_dummylist.svg~~350~~270~~

1. Move Window position one to the left.

Note: The dummylist diagram is used by the FOR command to scan through the argument list of the FOR to the body of the FOR.

---

a10t044: deviceparameters

deviceparameters

See deviceparameters for formal definition.

~~svg~~/image/AnnoStd/1984_trans_deviceparameters.svg~~400~~650~~

1. Null string ("") Right Arrow A.
2. Replace the n-tuple in A by the n + 1-tuple (A,Result).
3. If there exists a default value for this parameter of the device named in Devnam, replace the n-tuple in A by the n + 1-tuple (A,d), where d is the default value. Otherwise append a comma (",") to A to hold this parameter's position.
4. A Right Arrow Result.
5. Return the 1-tuple in Result.

---

a10t045: checkforparameters

checkforparameters

~~svg~~/image/AnnoStd/1984_trans_checkforparameters.svg~~410~~750~~

1. PUT Window position on Stack.
   One ("1") Right Arrow A.
2. A + 1 Right Arrow A.
3. A - 1 Right Arrow A.
   If A < 1, trap execution.
4. If A > 1, trap execution.
5. GET Window position from Stack.
   Link Window to retrieved Window position.

Note: The checkforparameters diagram is used by the deviceparameters diagram and the jobparmeters diagram to decide whether or not there are any parameters present.

---

a10t046: jobparameters

jobparameters

See jobparameters for formal definition.

~~svg~~/image/AnnoStd/1984_trans_jobparameters.svg~~400~~650~~

1. Null string ("") Right Arrow A.
2. Replace the n-tuple in A by the n + 1-tuple (A,Result).
3. A Right Arrow Result.
4. Return the 1-tuple in Result.

---

a10t047: nref

nref

See nref for formal definition.

~~svg~~/image/AnnoStd/1984_trans_nref.svg~~400~~650~~

1. Null string ("") Right Arrow A.
2. Circumflex ("^") Right Arrow A.

3. Concatenate (A, Result) Right Arrow A.
4. PUT Nameind on the Stack.
   False Right Arrow Nameind.
5. Replace the n-tuple in A with the n + 1-tuple (A, Result).
6. GET Nameind from the Stack.
7. GET previous level's Nameind and Window position from the Stack. GET the indirect name off the Stack.
   Link window to retrieved Window position.
8. A Right Arrow Result.

Note: The result of calling nref is an n-tuple of values, where the first value is the name, possible preceded by a "^", and subsequent values (if any) are subscripts.

---

a10t048: setpiece

setpiece

See setpiee for formal definition.

~~svg~~/image/AnnoStd/1984_trans_setpiece.svg~~750~~950~~

1. Result Right Arrow A.
2. Result Right Arrow B.
3. One ("1") Right Arrow C.
4. Result Right Arrow C.
5. C Right Arrow D.
6. Result Right Arrow D.
7. Place the 4-tuple (A;B;C;D) Right Arrow Result.

Note: The result returned by setpiece is a 4-tuple, where the first element is the global or local variable selected, the second element is the field delimiter to use for piece selection and replacement, and the third and fourth elements are the beginning and ending fields to be replaced.

Note: In the setpiece diagram, the corresponding lower-case letter is considered equivalent to the upper-case letter shown for the setpiece name.

---

a10t049: numexpr

numexpr

See numexpr for formal definition.

~~svg~~/image/AnnoStd/1984_trans_numexpr.svg~~75~~150~~

1. Result Right Arrow A.
   Take the numeric interpretation of the value in A.
   That is, apply the rules given in Section 3.2.5 of MDC/28 to A.
   A Right Arrow Result.

---

a10t050: intexpr

intexpr

See intexpr for formal definition.

~~svg~~/image/AnnoStd/1984_trans_intexpr.svg~~75~~150~~

1. Result Right Arrow A.
   Take the numeric interpretation of the value in A.
   That is, apply the rules given in Section 3.2.5 of MDC/28 to A.

Remove any fraction.
If the result is the null string (""), or "-", 0 Right Arrow A.
A Right Arrow Result.

---

a10t051: tvexpr

tvexpr

See tvexpr for formal definition.

~~svg~~/image/AnnoStd/1984_trans_tvexpr.svg~~75~~150~~

1. Result Right Arrow A.
   Take the numeric interpretation of the value in A.
   That is, apply the rules given in Section 3.2.5 of MDC/28 to A.
   If A ≠ 0, then 1 Right Arrow A.
   A Right Arrow Result.

---

a10t052: expr

expr

See expr for formal definition.

~~svg~~/image/AnnoStd/1984_trans_expr.svg~~350~~450~~

1. Result Right Arrow A.
2. Operator (Result) Right Arrow B.
3. Operator (not Result) Right Arrow B.
4. Operator (not ?) Right Arrow B.
5. Operator (?) Right Arrow B.
6. Apply the binary operator in B to A as left operand and Result as right operand, placing the resulting value in A.
7. Apply the pattern in Result to A, placing the resulting truth value in A.
   If the operator in B is not ?, logically complement A.
8. A Right Arrow Result.

---

a10t053: binaryop

binaryop

See binaryop for formal definition.

~~svg~~/image/AnnoStd/1984_trans_binaryop.svg~~650~~350~~

1. Char Right Arrow Result.

---

a10t054: truthop

truthop

See truthop for formal definition.

~~svg~~/image/AnnoStd/1984_trans_truthop.svg~~650~~350~~

1. Char Right Arrow Result.

---

a10t055: pattern

pattern

See pattern for formal definition.

~~svg~~/image/AnnoStd/1984_trans_pattern.svg~~370~~750~~

1. Result Right Arrow A.
2. If Result is not the null string (""), replace the n-tuple A by the n + 1-tuple (A, Result).
3. GET previous level's Nameind and Window position from the Stack.
   GET the indirect name off the Stack.
   Link window to retrieved Window position.
4. A Right Arrow Result.

Note: The value of pattern is an n-tuple of patatoms. It may be an 0-tuple.

---

a10t056: patatom

patatom

See patatom for formal definition.

~~svg~~/image/AnnoStd/1984_trans_patatom.svg~~330~~750~~

1. Result Right Arrow A.
2. Zero ("0") Right Arrow A.
3. Minus one ("–1") Right Arrow B.
4. Result Right Arrow B.
5. A Right Arrow B.
6. Result Right Arrow C.
7. Result as "literal" C.
8. Concatenate (C,Result) Right Arrow C.
9. If B is not minus one and B<A, trap execution.
   If A is not zero and B is net zero, place the 3-tuple (A,B,C) Right Arrow Result.
   Otherwise, null string ("") Right Arrow Result.

Note: The Result returned by patatom is a 3-tuple, where the value of the first element is either zero or the value of the first intlit, the value of the second element is either –1 (indefinite), or the value of the second intlit, and the value of the third element is either the result from strlit as "literal", or a string of patcodes. If the specified or implied value of both intlit is zero, patatom returns the null string.

---

a10t057: patcode

patcode

See patcode for formal definition.

~~svg~~/image/AnnoStd/1984_trans_patcode.svg~~650~~350~~

1. Char Right Arrow Result.

Note: In the patcode diagram, the corresponding lower-case letter is considered equivalent to the upper-case letter shown.

---

a10t058: expratom

expratom

See expratom for formal definition.

~~svg~~/image/AnnoStd/1984_trans_expratom.svg~~500~~400~~

1. Null string ("") Right Arrow A.
2. Concatenate (A,Result) Right Arrow A.
3. Search the set of local or global variables for the one with the name contained in Result.
   If the search is successful, place the value of the variable into Result.
   Then do action 5.
   If the search is not successful, trap execution.
4. If Nameind is True, trap execution.
   Otherwise, do action 5.
5. Apply the unary operators in A (if any) to the value in Result, in a right-to-left order.
   Place the result of this application into Result.

---

a10t059: unaryop

unaryop

See unaryop for formal definition.

~~svg~~/image/AnnoStd/1984_trans_unaryop.svg~~250~~350~~

1. Char Right Arrow Result.

---

a10t060: glvn

glvn

See glvn for formal definition.

~~svg~~/image/AnnoStd/1984_trans_glvn.svg~~500~~750~~

1. Set Setsw = False.
2. Result Right Arrow A.
3. Perform the following operations in order:
   a. If the first value of the n-tuple in A is the naked symbol along ("^"), substitute the value of the naked indicator for it.
      For example, if A is the n-tuple (^, s_1, s_2, ..., s_{n-1}) and the naked indicator is the m-tuple (^name, v_1, v_2, ..., v_{m-1}) and then A becomes the m - 1 + n-tuple (^name, v_1, v_2, ..., v_{m-1}, s_1, s_2, ..., s_{n-1}).
   b. The value in A is now an n-tuple (^name, x_1, x_2, ..., x_{n-1}).
      If A is a 1-tuple (no subscripts), make the naked indicator undefined.
      Otherwise, replace the naked indicator with the n - 1-tuple (^name, x_1, x_2, ..., x_{n-1}).
4. A Right Arrow Result.

---

a10t061: lvn

lvn

See lvn for formal definition.

~~svg~~/image/AnnoStd/1984_trans_lvn.svg~~550~~550~~

1. Result Right Arrow A.
2. PUT Nameind on the Stack.
   False Right Arrow Nameind.
3. Replace the n-tuple in A with the n + 1-tuple (A,Result).
4. GET Nameind from the Stack.
5. GET previous level's Nameind and Window position from the Stack.
   GET the indirect name off the Stack.

Link window to retrieved Window position.
6. A Right Arrow Result.

Note: The result of calling lvn is an n-tuple of values, where the first value is the name and subsequent values (if any) are subscripts.

---

a10t062: gvn

gvn

See gvn for formal definition.

~~svg~~/image/AnnoStd/1984_trans_gvn.svg~~550~~750~~

1. Circumflex ("^") Right Arrow A.
2. Concatenate (A, Result) Right Arrow A.
3. PUT Nameind on the Stack.
   False Right Arrow Nameind.
4. Replace the n-tuple in A with the n + 1-tuple (A, Result).
5. GET Nameind from the Stack.
6. GET previous level's Nameind and Window position from the Stack.
   GET the indirect name off the Stack.
   Link window to retrieved Window position.
7. A Right Arrow Result.

Note: The result of calling gvn is an n-tuple of values, where the first value is either the global name preceded by a "^", or the "^" alone, and subsequent values (if any) are subscripts.

---

a10t063: numlit

numlit

See numlit for formal definition.

~~svg~~/image/AnnoStd/1984_trans_numlit.svg~~420~~550~~

1. Period (".") Right Arrow A.
2. Remove leading zeros from Result.
   If Result > 0, Result Right Arrow A.
   Otherwise, zero ("0") Right Arrow A.
3. Concatenate (A, ".") Right Arrow A.
4. Remove trailing zeros from Result.
   Concatenate (A, Result) Right Arrow A.
5. Concatenate (A, "E") Right Arrow A.
6. Concatenate (A, "-") Right Arrow A.
7. Remove leading zeros from Result.
   Concatenate (A, Result) Right Arrow A.
8. Convert A to a numeric data value, using the algorithm defined in Section 3.2.4.2 of MDC/28.
   Place A Right Arrow Result.

---

a10t064: strlit

strlit

See strlit for formal definition.

~~svg~~/image/AnnoStd/1984_trans_strlit.svg~~255~~350~~

1. Null string ("") Right Arrow A.
2. Concatenate (A,Char) Right Arrow A.
3. Concatenate (A,""") Right Arrow A.
4. A Right Arrow Result.

---

a10t065: function

function

See function for formal definition.

~~svg~~/image/AnnoStd/1984_trans_function.svg~~1220~~2275~~

Note: In the function diagram, the corresponding lower-case letter is considered equivalent to the upper-case letter shown for function and special variable name.

1. Place present value of $HOROLOG (date and time) into Result.
2. Place present value of $IO (current I/O device) into Result.
3. Place present value of $JOB (number of this process) into Result.
4. Place present value of $STORAGE (remaining available storage) into Result.
5. Place present value of $TEST (Ifswitch) into Result.
6. Place present value of $X (horizontal cursor position of $IO device) into Result.
7. Place present value of $Y (vertical cursor position of $IO device) into Result.

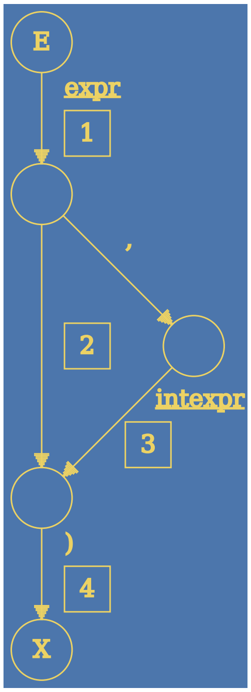
---

a10t066: $ASCII

$ASCII function

See $ASCII for formal definition.

1. Result Right Arrow A.
2. One ("1") Right Arrow B.
3. Result Right Arrow B.
4. If A is the null string, place –1 into Result.
   Otherwise, place the integer n associated with the character $E(A,B) into Result such that $A($C(n))=n.



---

a10t067: $CHAR

$CHAR function

See $Char for formal definition.

~~svg~~/image/AnnoStd/1984_trans_char.svg~~200~~450~~

1. Null string ("") Right Arrow A.
2. Result Right Arrow B.
3. If B > 127, trap execution. Otherwise, if B > –1, convert B to its ASCII character, place it into C, and concatenate (A,C) Right Arrow A.
4. A Right Arrow Result.

---

a10t068: $DATA

$DATA function

See $Data for formal definition.

~~svg~~/image/AnnoStd/1984_trans_data.svg~~100~~250~~

1. Result Right Arrow A.
Place characterization of the named variable in A into Result.

Note: The meaning of the value returned by the $DATA function is discussed in Section 3.2.8 of MDC/28.

---

a10t069: $EXTRACT

$EXTRACT function

See $Extract for formal definition.

~~svg~~/image/AnnoStd/1984_trans_extract.svg~~200~~650~~

1. Result Right Arrow A.
2. One ("1") Right Arrow B.
3. Result Right Arrow B.
4. B Right Arrow C.
5. Result Right Arrow C.
6. Extract from A the Bth through Cth characters and place into Result.

---

a10t070: $FIND

$FIND function

See $Find for formal definition.

~~svg~~/image/AnnoStd/1984_trans_find.svg~~150~~600~~

1. Result Right Arrow A.
2. Result Right Arrow B.
3. One ("1") Right Arrow C.
4. Result Right Arrow C.
5. Find the first occurrence of B in A, beginning at the Cth character of A.
   If B is contained in A, place n + 1 into Result, where n is the position in A of the last character in B.
   Otherwise, 0 Right Arrow Result.

---

a10t071: $JUSTIFY

$JUSTIFY function

See $Justify for formal definition.

~~svg~~/image/AnnoStd/1984_trans_justify.svg~~150~~600~~

1. Result Right Arrow A.
2. Result Right Arrow B.
3. 3. Right justify A in a field of B spaces and place into C.
4. If Result < 0, trap execution.
   Otherwise, perform the following operations in order:
   a. Take the numeric interpretation of A.
      That is, apply the rules given in Section 3.2.5 of Part I to A.
   b. If Result = 0, round the numeric value in A to an integer and remove the decimal point.
      Otherwise, round A to Result fraction digits (pad with trailing zeros if necessary; if the rounded value is between –1 and 1, place a zero to the left of the decimal point).
   c. Do action 3.
5. C Right Arrow Result.

---

a10t072: $LENGTH

$LENGTH function

See $Length for formal definition.

~~svg~~/image/AnnoStd/1984_trans_length.svg~~200~~450~~

1. Result Right Arrow A.
2. Place length of A in characters into Result.
3. Result Right Arrow B.
4. Place into Result the number plus one of nonoverlapping occurrences of B in A.

---

a10t073: $NEXT

$NEXT function

See $Next for formal definition.

~~svg~~/image/AnnoStd/1984_trans_next.svg~~100~~250~~

1. Result Right Arrow A.
   Check to see that A is at least a 2-tuple (that is, that the variable is subscripted).
   If not, generate an execution trap.
2. Find the next higher subscript of the variable named in A and place into Result.
   If no higher subscript exists, place –1 into Result.

Note: The details of action 2 are discussed in Section 3.2.8 of MDC/28.

---

a10t074: $ORDER

SORDER function

See $Order for formal definition.

~~svg~~/image/AnnoStd/1984_trans_order.svg~~100~~250~~

1. Result Right Arrow A.
   Check to see that A is at least a 2-tuple (that is, that the variable is subscripted).
   If not, generate an execution trap.

2. Find the next higher subscript of the variable named in A and place into Result.
   If no higher subscript exists, place the null string ("") into Result.

Note: The details of action 2 are discussed in Section 3.2.8 of Part I.

---

a10t075: $PIECE

$PIECE function

See $Piece for formal definition.

~~svg~~/image/AnnoStd/1984_trans_piece.svg~~155~~850~~

1. Result Right Arrow A.
2. Result Right Arrow B.
3. One ("1") Right Arrow C.
4. Result Right Arrow C.
5. C Right Arrow D.
6. Result Right Arrow D.
7. Place the C through D fields in A, delimited by B, into Result.

Note: The details of action 6 are described in Section 3.2.8 of Part I.

---

a10t076: $RANDOM

$RANDOM function

See $Random for formal definition.

~~svg~~/image/AnnoStd/1984_trans_random.svg~~100~~250~~

1. If Result < 1, trap execution.
   Otherwise, Result Right Arrow A.
2. Compute a "random" integer between zero and A - 1 inclusive, and place into Result.

---

a10t077: $SELECT

$SELECT function

See $Select for formal definition.

~~svg~~/image/AnnoStd/1984_trans_select.svg~~250~~850~~

1. (Privileged) Linact Right Arrow C,
   Comact Right Arrow D.
   Make B undefined.
2. Result Right Arrow A.
3. Set Comact = False.
4. Result Right Arrow B.
   Set Linact = False.
5. (Privileged) Set Comact = True.
6. (Privileged) C Right Arrow Linact,
   D Right Arrow Comact.
7. If B is undefined, trap execution.
   Otherwise, B Right Arrow Result.

Note: Linact and Comact are used only locally here to inhibit evaluation of the second expression in a False Boolean pair. However, syntax checking is performed.

---

a10t078: $TEXT

$TEXT function

See $Text for formal definition.

~~svg~~/image/AnnoStd/1984_trans_text.svg~~200~~250~~

1. Find the line in the present routine referenced by the lineref in Result.
   If no such line exists, place the null string ("") into C.
   Otherwise, line Right Arrow C.
2. If Result < 0, trap execution.
   Otherwise, Result Right Arrow B.
3. If B = 0, Present routine name Right Arrow C.
   Otherwise, find the Bth line of the present routine.
   If no such line exists, place the null string ("") into C.
   Otherwise, line Right Arrow C.
4. If C is not the null string (""), and C is not the Present routine name, replace the ls in C with one SP (space) and remove the eol.
   C Right Arrow Result.

---

a10t900: footnotes

Footnotes

Note 1

Conway, M. E., "Design of a Separable Transition-Diagram Compiler," Communications of the Association for Computing Machinery, 6:7 (July 1963), pp. 396–408.

Note 2

Conway, M. E., "Preliminary MUMPS Language Specification," MDC 1/3, Draft Proposal, Revised 7/12/74, MUMPS Development Committee.

Note 3

Wasserman, A. I. and Sherertz, D. D., "Implementation of the MUMPS Language Standard," MDC 2/3, 6/15/75, MUMPS Development Committee.

Privileged Action

This action is "Privileged"; it is always executed.

---

a200001: Portability Requirements

MUMPS LANGUAGE STANDARD
Part III: MUMPS Portability Requirements

1. Introduction

This document highlights, for the benefit of implementors and application programmers, aspects of the language that must be accorded special attention if MUMPS program transferability (i.e., portability of source code between various MUMPS 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; application programmers, on the other hand, may meet but must not exceed the limits guaranteed by this document. 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 MUMPS 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.

---

a202001: Names

2. Expression Elements
2.1 Names

The use of alpha 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.

---

a202003: Number of Local Variables

2.2 Local Variables
2.2.1 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 (Section 6).

---

a202004: Number of Subscripts

2.2.2 Number of subscripts

The number of subscripts in a local variable is limited in that, in a local array reference, the sum of the lengths of all the evaluated subscripts, plus the number of subscripts, plus the length of the local variable name must not exceed 63.

---

a202005: Values of Subscripts

2.2.3 Values of Subscripts

Local variable subscript values are nonempty strings which may only contain characters from the ASCII printable characters subset. The length of each subscript is limited to 31 characters. When the subscript value satisfies the definition of a numeric data value (Section 3.2.4.1 of the MUMPS Language Specification), it is further subject to the restrictions of number range given in Section 2.5, and to the set of nonnegative numbers only. 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 reference in the context of the $ORDER function, and the use of the value "–1" as the last subscript of a reference in the context of the $NEXT function.

---

a202006: Number of nodes

2.2.4 Number of Nodes

There is no explicit limit on the number of distinct nodes which are defined within local variable arrays. However, the limit on the number of local variables (Section 2.2.1) and the limit on the number of subscripts (Section 2.2.2) place an implicit limit on the number of distinct nodes which may be defined.

---

a202007: Scope of Local Variables

2.2.5 Scope of Local Variables

Local variables are unique to a process. All routines executed by a process share the same name space.

a202008: Number of Global Variables

2.3 Global Variables
2.3.1 Number of Global Variables

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

a202009: Number of Subscripts

2.3.2 Number of subscripts

The number of subscripts in a global variable is limited in that, in a global array reference, the sum of the lengths of all the evaluated subscripts, plus the number of subscripts, plus the length of the global variable name must not exceed 63. If a naked reference is used to specify the global array reference, the above restriction applies to the full reference to which the naked reference is expanded.

a202010: Values of Subscripts

2.3.3 Values of Subscripts

The restrictions imposed on the values of global variable subscripts are identical to those imposed on local variable subscripts (Section 2.2.3).

a202011: Number of Nodes

2.3.4 Number of Nodes

There is no limit on the number of distinct global variable nodes which are defined, since successive naked references may be used to access and/or create nodes at any depth within a global array.

a202012: Data Types

2.4 Data Types

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

a202013: Number Range

2.5 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.

The accuracy of any value used in arithmetic operations or in any context requiring a numeric interpretation is nine significant digits.

Programmers should exercise caution in the use of noninteger arithmetic. In general, arithmetic operations on noninterger 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.

a202014: Integers

2.6 Integers

The magnitude of the value resulting from an integer interpretation is limited by the accuracy of numeric values (Section 2.5). The values produced by integer valued operators and functions also fall within this range (see Section 3.2.5.1 of the MUMPS Language Specification for a precise definition of integer interpretation).

---

a202015: Character Strings

2.7 Character Strings

Character string length is limited to 255 characters. The characters permitted within character strings are those defined in the ASCII Standard (ANSI X3.4–1977).

---

a202016: Special Variables

2.8 Special Variables

The special variables $X and $Y are nonnegative integers (Section 2.6). The effect of incrementing $X and/or $Y past the maximum allowable integer value is undefined (for a description of the cases in which $X and $Y are incremented see the MUMPS Language Specification, Section 3.5.5).

---

a203001: Expressions (Nesting)

3. Expressions

3.1 Nesting of Expressions

The number of levels of nesting in expressions is not explicitly limited. The maximum string length does impose an implicit limit on this number (Section 2.7).

---

a203002: Results

3.2 Results

Any result, whether intermediate or final, which does not satisfy the constraints on character strings (Section 2.7) is erroneous. Furthermore, integer results are erroneous if they do not also satisfy the constraints on integers (Section 2.6).

---

a204001: Command Lines

4. Routines and Command Lines

4.1 Command Lines

A command line (line) must satisfy the constraints on character strings (Section 2.7). The length of a command line is determined as follows. Each character in the label (if present) counts as one character. The ls character counts as one character (note that command lines will therefore always be at least one character long). Each character following the is up to but not including the following eol counts as one character. The sum of the lengths of these three components (label, ls, and the command line proper) determines the length of the command line.

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 (Section 5).

---

a204002: Number of Command Lines

4.2 Number of Command Lines

There is no explicit limit on the number of command lines in a routine, subject to storage space restrictions (Section 6).

---

a204003: Number of Commands

4.3 Number of Commands

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

---

a204004: Labels

4.4 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 (Section 2.1).

---

a204005: Number of Labels

4.5 Number of Labels

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

a. A command line may have only one label.
b. No two lines may be labeled with equivalent (not uniquely distinguishable) labels.

---

a204006: Number of Routines

4.6 Number of Routines

There is no explicit limit on the number of routines. The number of routines is implicitly limited by the name length restriction (Section 2.1).

---

a207001: Indirection

5. Indirection

The value of the argument of indirection and the argument of the XECUTE command are subject to the constraints on character string length (Section 2.7). They are additionally restricted to the character set limitations of command lines (Section 4. 1).

---

a208001: Storage Space Restrictions

6. Storage Space Restrictions

MUMPS has traditionally been implemented on small to medium size computers using a scheme of fixed memory allocation, one fixed partition per user. It is recognized that more flexible storage allocation techniques can be applied and there is no intent to restrict implementations to use of the traditional techniques. Nevertheless, because partitioned memory implementations will continue to be important for some time, certain storage restrictions are required to permit program portability. These restrictions have been defined in terms of parameters which are implementation-independent and observable to the application programmer.

The storage restrictions on portable programs are expressed in the following rule. At any time during the execution of a process, routine size plus local variable storage size plus temporary result storage size must not exceed 4000 characters. Storage space for control purposes, device buffers, disc buffers, line buffers, etc. is not included in this count.

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 Section 4.1) plus two.

The size of local variable storage is the sum of the sizes of all the simultaneously defined local variables. The size of an unsubscripted local variable is the length of its name in characters plus the length of its value in characters, plus two. The size of a local array is the sum of the following.

a. The length of the name of the array.
b. Two 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)>10.

All subscripts and values are considered to be character strings for this purpose.

All intermediate results generated during the processes of expression evaluation, indirection, multiple SET command scanning, etc. require the use of temporary storage. At any given time, the amount of temporary storage required is the sum of the lengths of all simultaneously existing temporary results. All temporary results are maintained as strings of contiguous characters.

---

a209001: Process-Stack

7. Nesting

Each active DO, FOR, XECUTE, and indirection occurrence is counted as a level of nesting. Control storage provides for fifteen levels of nesting. The actual use of all these levels may be limited by storage restrictions (Section 6).

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 (Section 6).

---

a213001: Other Portability Requirements

8. 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 or a timeout cannot be expected to be exact. In particular, relying upon noninteger values in these situations can lead to unexpected results.

---

aa01001: charset M

Appendix A

(This Appendix is not a part of American National Standard ANSI/MDC X11.1–1984)

Appendix A
ASCII Character Set

Octal Code	Decimal Code	Character	Patcode
0	0	NUL	C,E
1	1	SOH	C,E
2	2	STX	C,E
3	3	ETC	C,E
4	4	EOT	C,E
5	5	ENQ	C,E
6	6	ACK	C,E
7	7	BELL	C,E
10	8	BS	C,E

11	9	HT		C,E
12	10	LF		C,E
13	11	VT		C,E
14	12	FF		C,E
15	13	CR		C,E
16	14	SO		C,E
17	15	SI		C,E
20	16	DLE		C,E
21	17	DC1		C,E
22	18	DC2		C,E
23	19	DC3		C,E
24	20	DC4		C,E
25	21	NAK		C,E
26	22	SYN		C,E
27	23	ETB		C,E
30	24	CAN		C,E
31	25	EM		C,E
32	26	SUB		C,E
33	27	ESC		C,E
314	28	FS		C,E
35	29	GS		C,E
36	30	RS		C,E
37	31	US		C,E
40	32	Space		P,E
41	33	!		P,E
42	34	"		P,E
43	35	#		P,E
44	36	$		P,E
45	37	%		P,E
46	38	&		P,E
47	39	'	(note: apostrophe)	P,E
50	40	(		P,E
51	41	)		P,E
52	42	*		P,E
53	43	+		P,E
54	44	,	(note: comma)	P,E
55	45	-	(note: hyphen)	P,E
56	46	.		P,E
57	47	/		P,E
60	48	0		N,E
61	49	1		N,E
62	50	2		N,E
63	51	3		N,E
64	52	4		N,E
65	53	5		N,E
66	54	6		N,E
67	55	7		N,E
70	56	8		N,E
71	57	9		N,E
72	58	:		P,E
73	59	;		P,E
74	60	<		P,E
75	61	=		P,E
76	62	>		P,E

77	63	?		P,E
100	64	@		P,E
101	65	A		A,U,E
102	66	B		A,U,E
103	67	C		A,U,E
104	68	D		A,U,E
105	69	E		A,U,E
106	70	F		A,U,E
107	71	G		A,U,E
110	72	H		A,U,E
111	73	I		A,U,E
112	74	J		A,U,E
113	75	K		A,U,E
114	76	L		A,U,E
115	77	M		A,U,E
116	78	N		A,U,E
117	79	0		A,U,E
120	80	P		A,U,E
121	81	Q		A,U,E
122	82	R		A,U,E
123	83	S		A,U,E
124	84	T		A,U,E
125	85	U		A,U,E
126	86	V		A,U,E
127	87	W		A,U,E
130	88	X		A,U,E
131	89	Y		A,U,E
132	90	Z		A,U,E
133	91	[		P,E
134	92	\		P,E
135	93	]		P,E
136	94	^		P,E
137	95	_	(note: underscore)	P,E
140	96	`		P,E
141	97	a		A,L,E
142	98	b		A,L,E
143	99	c		A,L,E
144	100	d		A,L,E
145	101	e		A,L,E
146	102	f		A,L,E
147	103	g		A,L,E
150	104	h		A,L,E
151	105	i		A,L,E
152	106	j		A,L,E
153	107	k		A,L,E
154	108	1		A,L,E
155	109	m		A,L,E
156	110	n		A,L,E
157	111	o		A,L,E
160	112	p		A,L,E
161	113	q		A,L,E
162	114	r		A,L,E
163	115	s		A,L,E
164	116	t		A,L,E

165	117	u	A,L,E
166	118	v	A,L,E
167	119	w	A,L,E
170	120	x	A,L,E
171	121	y	A,L,E
172	122	z	A,L,E
173	123	{	P,E
174	124	|	P,E
175	125	}	P,E
176	126	~	P,E
177	127	DEL	C,E