ANSI/MDC X11.1-1990

**American National Standard**
for Information Systems -
Programming Languages -

# MUMPS

*Sponsor*
**MUMPS Development Committee**

*Approved November 11, 1990*
**American National Standards Institute, Inc**

Abstract This standard contains a two-part description of various aspects of the MUMPS computer programming language. Part 1, MUMPS Language Specification, consists of a stylized English narrative definition of the MUMPS language. Part 2, 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.

# 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.

The American National Standards Institute does not develop standards and will in no circumstances give an interpretation of any American National Standard. Moreover, no person shall have the right or authority to issue an interpretation of an American National Standard in the name of the American National Standards Institute. Requests for interpretations should be addressed to the secretariat or sponsor whose name appears on the title page of this standard.

**CAUTION NOTICE**: This American National Standard may be revised or withdrawn at any time.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 Users' Group**
**4321 Hartwick Road, #100**
**College Park, MD. 20740**

Reproduced under license from The MUMPS Development Committee by the MUMPS Users' Group.



Printed in the United States of America

# Foreword

(This Foreword is not part of American National Standard ANSI/MDC X11.1-1990.)

MUMPS, an acronym for **M**assachusetts General Hospital **U**tility **M**ulti-**P**rogramming **S**ystem, 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. The standard was revised and approved again on November 15, 1984. 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 on March 20, 1989. Approval was granted on November 11, 1990.

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 MDC Secretariat, 4321 Hartwick Road, Suite 100, College Park, MD 20740.

The reader shall note that $TEXT is not entirely backwards compatible with previous versions of this standard. Specifically, the following phrase was removed from the description of the result of $TEXT, "*Is replaced by one SP* ". This change has the effect of preserving the indentation of command lines. Routines which expect $TEXT to remove all extraneous spaces before the first command in a line will have to be modified.

The reader shall note that $NEXT may be removed from the next revision of this standard. The use of $ORDER instead of $NEXT is strongly encouraged.

Consensus for approval of this standard as an American National Standard was achieved through the use of the Canvass Method. The following organizations represent the Canvass List.

| Organization | Organization |
|---|---|
| AICCP | Health Data Sciences Corp |
| American Library Assoc | Heffernan, Henry |
| ANSI X3 | IBM |
| Bell, Mitchell | IDX |
| Braughler, Denver | Indian Health Services |
| Bristol-Meyers Co | Interactive Software Products |
| Bull HN Info Sys Inc | Interactive Technology Inc |
| Center For Biostatistics | InterSystems Corp |
| Computerized Medical Systems | Isotech |
| Connections Group, Ltd | J Gerald McManus, Inc |
| Cornell Univ | Johns Hopkins Health Plan |
| Crusader Info Sys | Johnson Wax |
| Crystalline Creations, Inc | Kbase Systems |
| DataTree, Inc | Klein, James R |
| Davidson, Fred T | Lawrence Berkeley Lab |
| DI*Star Medical Systems Corp | M Systems Plus Inc |
| Digital Equipment Corp | Maryland Medical Lab |
| DMSSC | Mass General Hospital |
| Educational Systems Inc | Mass Inst of Technology |
| Elderkin, David | Med-Check Labs |
| Exchange Carriers Stds Assn | Metcalf, Roger |
| Fenner, Stanley | Meyn Consulting |
| Forrey, Arden | MGlobal |
| George Washington Univ Med Ctr | Micah Systems Inc |
| Greystone Tech Corp | Micronetics Design Corp |
| GSA - ADTS | Mitre Corp |
| Guide, Intl - American Management Systems | NIST |
| Health Care Mgmt Counselors | O'Gorman, Kevin |

Canvass List (cont.)
Organization

Omnicom Inc
Online Computer Systems Inc
Online Medical Networks
Pioneer Data Sys
Plus Five Computer Svcs
Polylogics Consulting
Pyramid Technology
Quality Data Services
Ream, Norman J
Ruh, Lawrence A
SAIC
Sakowitz Computer Lab
SBS-International
Scott White Clinic
Shared Medical Systems
Shefrin, Elliot A
Smith, Tony W
Smith-Kline Bio-Sciences Lab
St Joseph Health Sys
Texas Instruments
Thomas E Harris & Co
Todd, Michael L
Unisys Corp
Univ of Alabama
Univ of California - Davis
Univ of California - Santa Barbara
Univ of Missouri
Univ of Washington
US Dept of Defense
Veterans Administration - Bedford, MA
Veterans Administration - Birmingham, AL
Veterans Administration - Great Lakes Region
Veterans Administration - Jackson, MS
Veterans Administration - Lexington, KY
Veterans Administration - Washington, DC
Vogt, Michael C
Volkstorf, Charles S

# Contents

# Part 1: MUMPS Language Specification

## Table of Contents

# American National Standard for Information Systems - Programming Languages - MUMPS (Part 1: MUMPS Language Specification)

## Introduction

Part 1 consists of two sections that describe the MUMPS language. Section 1 describes the metalanguage used in the remainder of Part 1 for the static syntax. Section 2 describes the static syntax and overall semantics of the language. The distinction between "static" and "dynamic" syntax is as follows. The static syntax describes the sequence of characters in a routine as it appears on a tape in routine interchange or on a listing. The dynamic syntax describes the sequence of characters that would be encountered by an interpreter during execution of the routine. (There is no requirement that MUMPS actually be interpreted). The dynamic syntax takes into account transfers of control and values produced by indirection.

## 1 Static Syntax Metalanguage

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

| Operator | Meaning |
|---|---|
| ::= | definition |
| [ ] | option |
| \| \| | grouping |
| ... | optional indefinite repetition |
| L | list |
| V | value |
| SP | space |

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

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

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

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

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

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

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

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

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

where

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

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

# 2 Static Syntax and Semantics

### 2.1 Basic Alphabet

The routine, which is the object whose static syntax is being described in Section 2, is a string made up of the following 98 ASCII symbols.

> The 95 printable characters, including the space character represented as SP, and also,
> the carriage-return character represented as CR,
> the line-feed character represented as LF,
> the form-feed character represented as FF.

See 2.4 for the definition of routine.

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

```
graphic  ::= any of the class of 95 ASCII printable characters, including
             SP.

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

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.
```

### 2.2 Expression Atom expratom

The expression, expr, is the syntactic element which denotes the execution of a value-producing calculation; it is defined in 2.3. The expression atom, expratom, is the basic value-denoting object of which expressions are built; it is defined here.

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

See 2.2.2.1 for the definition of lvn. See 2.2.2.2 for the definition of gvn.

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

See 2.2.6 for the definition of svn. See 2.2.7 for the definition of function. See 2.2.10 for the definition of exfunc. See 2.2.9 for the definition of exvar. See 2.2.3 for the definition of numlit. See 2.2.5 for the definition of strlit. See 2.3 for the definition of expr.

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

#### 2.2.1 Name name

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

See 2.1 for the definition of alpha and digit.

#### 2.2.2 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 generally available to all routines executed within the process' lifetime. MUMPS does include certain constructs, the NEW command and parameter passing, which limit the availability of certain variables to specific routines or parts of routines. See 2.2.2.3 for a further discussion of variables and variable environments.

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.

##### 2.2.2.1 Local Variable Name lvn

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

See 2.2 for the definition of expratom. See section 1 for the definition of V.

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

See 2.2.1 for the definition of name. See 2.3 for the definition of expr. See section 1 for the definition of L.

```
lnamind   ::=  rexpratom V lvn
```

See section 1 for the definition of V.

```
                | rlvn     |
rexpratom ::=   | rqvn     |
                | expritem |
```

See 2.2.2.2 for the definition of rgvn. See 2.2 for the definition of expritem.

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

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

#### 2.2.2.2 Global Variable Name gvn

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

See 2.2 for the definition of expratom. See section 1 for the definition of V.

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

See 2.3 for the definition of expr. See 2.2.1 for the definition of name. See section 1 for the definition of L.

```
gnamind  ::=  | rexpratom V gvn       |
```

See section 1 for the definition of V.

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

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

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

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

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

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

results in a global reference of the form

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

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

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

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

See 2.2.2.1 for the definition of lvn.

#### 2.2.2.3 Variable Handling

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

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

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

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

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

The value of a MUMPS variable may be described by a relationship between two structures: the NAME-TABLE and the VALUE-TABLE. (In reality, at least two such table sets are required, one pair per executing process for process-specific local variables and one pair for system-wide global variables.) Since the value association process is the same for both types of variables, and since issues of scoping due to parameter passing or nested environments apply only to local variables, the discussion that follows will address only

local variable value association. It should be noted, however, that while the overall structures of the table sets are the same, there are two major differences in the way the sets are used. First, the global variable tables are shared. This means that any operations on the global tables, e.g., SET or KILL, by one process, affect the tables for all processes. Second, since scoping issues of parameter passing and the NEW command are not applicable to global variables, there is always a one-to-one relationship between entries in the global NAME-TABLE (variable names) and entries in the global VALUE-TABLE (values).

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

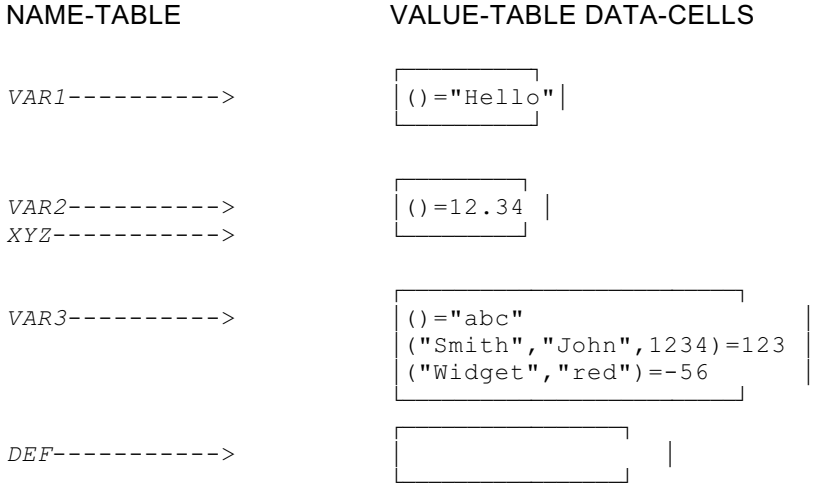
The NAME-TABLE entries contain every non-subscripted variable or array name (name) known, or accessible, by the MUMPS process in the current environment. The VALUE-TABLE DATA-CELLs contain the set of tuples that represent all variables currently having data-values for the process. Every name (entry) in the NAME-TABLE refers (*points*) to exactly one DATA-CELL, and every entry contains a unique name. Several NAME-TABLE entries (names) can refer to the same DATA-CELL, however, and thus there is a many-to-one relationship between (all) NAME-TABLE entries and DATA-CELLs. A name is said to be *bound* to its corresponding DATA-CELL through the *pointer* in the NAME-TABLE entry. Thus the pointer is used to represent the correspondence and the phrase *change the pointer* is the equivalent to saying *change the correspondence so that a name now corresponds to a possible different DATA-CELL (value)*. NAME-TABLE entries are also placed in the PROCESS-STACK (see 2.2.2.4).

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

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

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

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




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

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

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

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

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

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

When a variable is deleted (*killed*):

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

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

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

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

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

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

#### 2.2.2.4 Variable Contexts

The organization of multiple variable contexts requires the use of a PROCESS-STACK. This is a simple push-down stack, or last-in-first-out (LIFO) list, used to save and restore items which control the execution flow or variable environment. Five types of items, or frames, will be placed on the PROCESS-STACK, DO frames, exfunc frames, exvar frames, NEW frames, and parameter frames:

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

b. Exfunc and exvar frames contain saved values of $T, the execution level, and the execution location.

c. NEW frames contain a NEW argument (newargument) and a set of NAME-TABLE entries.

d. Parameter frames contain a formallist and a set of NAME-TABLE entries.

### 2.2.3 Numeric Literal numlit

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

```
intlit  ::=  digit ...
```

See 2.1 for the definition of digit.

The numeric literal numlit is defined as follows.

```
numlit  ::=  mant [ exp ]

mant    ::=  | intlit [. intlit] |
             | . intlit           |
```

```
exp      ::=  E [ + | - ] intlit
```

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

#### 2.2.3.1 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 noninteger 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.

#### 2.2.3.2 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".

#### 2.2.4 Numeric Interpretation of Data

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

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

Consider the argument to be the string *S*.

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

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

b. - + *T* → - *T*

c. - - *T* → *T*

Second, apply one of the following, as appropriate.

a. If the leftmost character of *S* is not "-", form the longest head of *S* which satisfies the syntax description of numlit. Then apply the algorithm of 2.2.3.2 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 2.3 for the definition of expr.

##### 2.2.4.1 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 2.3 for the definition of expr.

#### 2.2.4.2 Truth-Value Interpretation

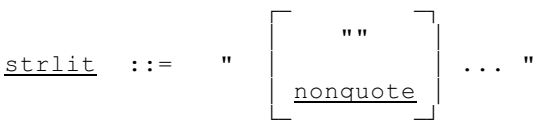
The truth-value interpretation is a mapping from the set of all data values onto the two integer values 0 (false) and 1 (true), described by the following algorithm. Take the numeric interpretation. If the result is not "0", make it "1".

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

```
tvexpr  :: = expr
```

See 2.3 for the definition of expr.

### 2.2.5 String Literal strlit




See 2.1 for the definition of nonquote.

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

### 2.2.6 Intrinsic Special Variable Name svn

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

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

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

The formal definition of the syntax of svn is a choice from among all of the individual svn syntax definitions of 2.2.6.

```
                | syntax of $HOROLOG intrinsic special variable |
                | syntax of $IO intrinsic special variable       |
svn ::=         |                       .                        |
                |                       .                        |
                |                       .                        |
                | syntax of $Y intrinsic special variable        |
```

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

| Syntax | Definition |
|---|---|
| $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. |
| $I[O] | $I identifies the current I/O device (see 2.6.2 and 2.6.18). |
| $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. |
| $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. |
| $T[EST] | $T contains the truth value computed from the execution of an IF command containing an argument, or an OPEN, LOCK, JOB, or READ command with a timeout (see 2.2.9, 2.2.10, and 2.6.3). |
| $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 2.5.5 and 2.6.18).

$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 2.5.5 and 2.6.18).

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

### 2.2.7 Intrinsic Functions function

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

```
A[SCII]
C[HAR]
D[ATA]
E[XTRACT]
F[IND]
FN[UMBER]
G[ET]
J[USTIFY]
L[ENGTH]
N[EXT]
O[RDER]
P[IECE]
Q[UERY]
R[ANDOM]
S[ELECT]
T[EXT]
TR[ANSLATE]
V[IEW]
Z[unspecified]
```

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

The formal definition of the syntax of function is a choice from among all of the individual function syntax definitions of 2.2.7

```
                | syntax of $ASCII function |
                | syntax of $CHAR function  |
                |            .              |
function ::=    |            .              |
                |            .              |
                | syntax of $VIEW function  |
```

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

#### 2.2.7.1 $ASCII

$A[SCII] ( expr )

See 2.3 for the definition of expr.

This form produces an integer value as follows:

a. -1 if the value of expr is the empty string.
b. Otherwise, an integer *n* associated with the leftmost character of the value of expr, such that $A($C(*n*)) = *n*.

$A[SCII] ( expr , intexpr )

See 2.3 for the definition of expr. See 2.2.4.1 for the definition of intexpr.

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

#### 2.2.7.2 $CHAR

$C[HAR] ( L intexpr )

See 2.2.4.1 for the definition of intexpr. See section 1 for the definition of L.

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

#### 2.2.7.3 $DATA

$D[ATA] ( glvn )

See 2.2.2.2 for the definition of glvn.

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

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

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

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

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

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

#### 2.2.7.4 $EXTRACT

$E[XTRACT] ( expr )

See 2.3 for the definition of expr.

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

$E[XTRACT] ( expr , intexpr )

See 2.3 for the definition of expr. See 2.2.4.1 for the definition of intexpr.

Let *s* be the value of expr, and let *m* be the integer value of intexpr. $E(*s*,*m*) returns the *m*th 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 , $\text{intexpr}_1$ , $\text{intexpr}_2$ )

See 2.3 for the definition of expr. See 2.2.4.1 for the definition of intexpr.

Let *n* be the integer value of $\text{intexpr}_2$. $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 2.3.5, $E(*s*,*m*,*n*) = $E(*s*,*m*)_$E(*s*,*m*+1)_..._$E(*s*,*m*+(*n*-*m*)).

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

#### 2.2.7.5 $FIND

$F[IND] ( $\text{expr}_1$ , $\text{expr}_2$ )

See 2.3 for the definition of expr.

This form searches for the leftmost occurrence of the value of $\text{expr}_2$ in the value of $\text{expr}_1$. 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 $\text{expr}_2$ in $\text{expr}_1$. In particular, if the value of $\text{expr}_2$ is empty, $F returns 1.

$F[IND] ( $\text{expr}_1$ , $\text{expr}_2$ , intexpr )

See 2.3 for the definition of expr. See 2.2.4.1 for the definition of intexpr.

Let *a* be the value of $\text{expr}_1$, let *b* be the value of $\text{expr}_2$, and let *m* be the value of intexpr. $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 0; otherwise, $F(*a*,*b*,*m*) = *p* + max(*m*,1) - 1.

#### 2.2.7.6 $FNUMBER

$FN[UMBER] ( numexpr , fncodexpr )

See 2.2.4 for the definition of numexpr.

```
fncodexpr ::= expr V fncode
```

See 2.3 for the definition of expr. See section 1 for the definition of V.

```
fncode     ::= fncodatom ...

               |P|
               |T|
fncodatom  ::= |,|      (note, comma)
               |+|
               |-|      (note, hyphen)
```

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

| fncodatom | Action |
|---|---|
| P | Represent negative numexpr values in parentheses. Let *A* be the absolute value of numexpr. Use of fncode "P" will result in the following:<br>1. If numexpr < 0, the result will be "("_*A*_")".<br>2. If numexpr '< 0, the result will be " "_*A*_" ". |
| T | Represent numexpr with a trailing rather than a leading "+" or "-" sign. Note: if sign suppression is in force (either by default on positive values, or by design using the "-" fncodatom), use of fncode "T" will result in a trailing space character. |
| , | Insert comma delimiters every third position to the left of the decimal (present or assumed) within numexpr. Note: no comma shall be inserted which would result in a leading comma character. |
| + | Force a plus sign ("+") on positive values of numexpr. Position of the "+" (leading or trailing) is dependent on whether or not fncodatom of "T" is specified. |
| - | Suppress the negative sign "-" on negative values of numexpr. |

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

More than one occurrence of a particular fncodatom within a single fncode is identical to a single occurrence of that fncodatom. Erroneous conditions are produced when a fncodatom "P" is present with any of the sign suppression or sign placement fncodatoms ("T+-").

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

See 2.2.4 for the definition of numexpr. See 2.2.4.1 for the definition of intexpr.

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

#### 2.2.7.7 $GET

$G[ET] ( glvn )

See 2.2.2.2 for the definition of glvn.

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

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

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

#### 2.2.7.8 $JUSTIFY

$J[USTIFY] ( expr , intexpr )

See 2.3 for the definition of expr. See 2.2.4.1 for the definition of intexpr.

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

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

b. Otherwise, the value returned is *S*(*n*-*m*) concatenated with $expr_1$, where *S*(*x*) is a string of *x* spaces.

$J[USTIFY] ( numexpr , $intexpr_1$ , $intexpr_2$ )

See 2.2.4 for the definition of numexpr. See 2.2.4.1 for the definition of intexpr.

This form returns an edited form of the number numexpr. Let *r* be the value of numexpr after rounding to $intexpr_2$ fraction digits, including possible trailing zeros. (If $intexpr_2$ is the value 0, *r* contains no decimal point.) The value returned is $J(*r*, $intexpr_1$). Note that if -1 < numexpr < 1, the result of $J does have a zero to the left of the decimal point. Negative values of $intexpr_2$ are reserved for future extensions of the $JUSTIFY function.

#### 2.2.7.9 $LENGTH

$L[ENGTH] ( expr )

See 2.3 for the definition of expr.

This form returns an integer which is the number of characters in the value of expr. If the value of expr is

the empty string, $L(expr) returns the value 0.

$L[ENGTH] ( $expr_1$ , $expr_2$ )

See 2.3 for the definition of expr.

This form returns the number plus one of nonoverlapping occurrences of $expr_2$ in $expr_1$. If the value of $expr_2$ is the empty string, then $L returns the value 0.

#### 2.2.7.10 $NEXT[1]

$N[EXT] ( glvn )

See 2.2.2.2 for the definition of glvn.

This form 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 (see 2.2.3.1), 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*:

[1]$NEXT may not be included in the next version of the X11 MUMPS Standard.

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.

#### 2.2.7.11 $ORDER

$O[RDER] ( glvn )

See 2.2.2.2 for the definition of glvn.

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

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

When $t$ follows $s$ in the ordering sequence, CO($s$,$t$) returns $t$.
Otherwise, CO($s$,$t$) returns $s$.

Let $m$ and $n$ be strings satisfying the definition of numeric data values (see 2.2.3.1), 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($s_1$, $s_2$, ..., $s_n$) where $s_n$ may be the empty string. Let $A$ be the set of subscripts that follow $s_n$. That is, for all $s$ in $A$:

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

Then $O(Name($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.

#### 2.2.7.12 $PIECE

$P[IECE] ( $expr_1$ , $expr_2$ )

See 2.3 for the definition of expr.

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

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

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

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

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

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

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

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

Let *s* be the value of $\underline{\text{expr}}_1$, and let *d* be the value of $\underline{\text{expr}}_2$. $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*,0,NF(*s*,*d*,1) - $L(*d*) - 1).

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

See 2.3 for the definition of expr. See 2.2.4.1 for the definition of intexpr.

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

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

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

See 2.3 for the definition of expr. See 2.2.4.1 for the definition of intexpr.

Let *n* be the integer value of $\underline{\text{intexpr}}_2$. $P(*s*,*d*,*m*,*n*) returns the substring of *s* bounded on the left but not including the *m-1*th occurrence of *d* in *s*, and bounded on the right but not including the *n*th occurrence of *d* in *s*.

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

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

#### 2.2.7.13 $QUERY

$Q[UERY] ( glvn )

See 2.2.2.2 for the definition of glvn.

Let $\underline{\text{glvn}}_1$ = ^A($i_1$, $i_2$, ..., $i_p$) and $\underline{\text{glvn}}_2$ = ^A($j_1$, $j_2$, ..., $j_q$)

Then a collating relation exists between glvn$_1$ and glvn$_2$. Glvn$_2$ is said to follow glvn$_1$ when for any pair ($i_k$, $j_k$) when:

a. $p < q$ and $i_k = j_k$ for all $k$ in the range ($0 < k \text{ '> } p$).

or

b. $p = 0$ and $q > 0$.

or

c. $k > 0$ and $k \text{ '> } \min(p, q)$ and
there exists some $i_k \text{ '= } j_k$ and
no $n$ exists with $0 < n < k$ and $i_n \text{ '= } j_n$ and
the function $CO(i_k, j_k)$, as defined in 2.2.7 (definition of $ORDER), is equal to $j_k$.
In less formal terms, when the first index that is different, collates in glvn$_2$ after the corresponding one in glvn$_1$.

For the purpose of this discussion a function CQ(glvn$_1$, glvn$_2$) is defined that would yield glvn$_2$ when, according to the above definition, glvn$_2$ would be said to follow glvn$_1$.

```
namevalue ::= expr
```

See 2.3 for the definition of expr.

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

a. The glvn is not a naked reference.

b. Each subscript whose value has the form of a number as defined in 2.2.3.1 appears as a numlit, spelled as its numeric interpretation.

c. Each subscript whose value does not have the form of a number as defined in 2.2.3.1 appears as a sublit, defined as follows:

```
sublit ::= " |     ""      |   "
           | subnonquote |...
```

where subnonquote is defined as follows:

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

Then the value of the function $QUERY can be defined as follows:

the value of $QUERY(glvn$_1$) is a namevalue that conforms to glvn$_2$ if and only if:

CQ (glvn$_1$, glvn$_2$) = glvn$_2$

and no glvn$_3$ exists so that

CQ (glvn$_1$, glvn$_3$) = glvn$_3$

and

CQ (glvn$_3$, glvn$_2$) = glvn$_2$

the value of $QUERY(glvn$_1$) will be the empty string when no glvn$_2$ exists so that

CQ (glvn$_1$, glvn$_2$) = glvn$_2$

#### 2.2.7.14 $RANDOM

$R[ANDOM] ( intexpr )

See 2.2.4.1 for the definition of intexpr.

This form returns a random or pseudo-random integer uniformly distributed in the closed interval [0, intexpr - 1]. If the value of intexpr is less than 1, an error will occur.

#### 2.2.7.15 $SELECT

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

See 2.2.4.2 for the definition of tvexpr. See 2.3 for the definition of expr. See section 1 for the definition of L.

This form returns the value of the leftmost expr whose corresponding tvexpr is true. The process of evaluation consists of evaluating the tvexprs, one at a time in left-to-right order, until the first one is found whose value is true. The expr corresponding to this tvexpr (and no other) is evaluated and this value is made the value of $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.

#### 2.2.7.16 $TEXT

```
          | + intexpr  |
$T[EXT] ( |            | )
          |  entryref  |
```

See 2.2.4.1 for the definition of intexpr. See 2.5.7 for the definition of entryref.

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

If the argument of $T is an entryref, the line denoted by the entryref is specified. If entryref does not contain dlabel then the line denoted is the first line of the routine. If the argument is + 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.

#### 2.2.7.17 $TRANSLATE

$TR[ANSLATE] ( $\underline{expr}_1$ , $\underline{expr}_2$ )

See 2.3 for the definition of expr.

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

$TR[ANSLATE] ( $\underline{expr}_1$ , $\underline{expr}_2$ , $\underline{expr}_3$ )

See 2.3 for the definition of expr.

Let *s* be the value of $\underline{expr}_1$, $TR($\underline{expr}_1$,$\underline{expr}_2$,$\underline{expr}_3$) returns an edited form of *s* in which all characters in *s* which are found in $\underline{expr}_2$ are replaced by the positionally corresponding character in $\underline{expr}_3$. If a character in *s* appears more than once in $\underline{expr}_2$ the first (leftmost) occurrence is used to positionally locate the translation.

Translation is performed once for each character in *s*. Characters which are in *s* that are not in $\underline{expr}_2$ remain unchanged. Characters in $\underline{expr}_2$ which have no corresponding character in $\underline{expr}_3$ are deleted from *s* (this is the case when $\underline{expr}_3$ is shorter than $\underline{expr}_2$).

Note: If the value of $\underline{expr}_2$ is the empty string, no translation is performed and *s* is returned unchanged.

#### 2.2.7.18 $VIEW

$V[IEW] ( unspecified )

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

#### 2.2.7.19 $Z

$Z[unspecified] ( unspecified )

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

### 2.2.8 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.

#### 2.2.9 Extrinsic Special Variable

```
exvar ::= $$ labelref
```

See 2.5.7 for the definition of labelref.

Extrinsic special variables are denoted by the prefix $$ followed by a labelref. Extrinsic special variables invoke a MUMPS subroutine to return a value. When an extrinsic special variable is executed, the current value of $T, the current execution level, and the current execution location are saved in an exvar frame on the PROCESS-STACK.

Execution continues at the first command of the formalline specified by the labelref. Execution of an exvar to a levelline is erroneous.

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

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

```
$$x()
```

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

#### 2.2.10 Extrinsic Function

```
exfunc ::= $$ labelref actuallist
```

See 2.5.7 for the definition of labelref. See 2.5.9 for the definition of actuallist.

Extrinsic functions are denoted by the prefix $$ followed by a labelref followed by an actuallist of parameters. Extrinsic functions invoke a MUMPS subroutine to return a value. When an extrinsic function is executed, the current value of $T, the current execution level, and the current execution location are saved in an exfunc frame on the PROCESS-STACK. The actuallist parameters are then processed as described in 2.5.9.

Execution continues at the first command of the formalline specified by the labelref. This formalline must contain a formallist in which the number of names is greater than or equal to the number of names in the actuallist. Execution of an exfunc to a levelline is erroneous.

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

### 2.3 Expressions expr

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

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

See 2.2 for the definition of expratom.

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

See 2.2 for the definition of expratom. See 2.3.3 for the definition of pattern.

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

truthop   ::=     | relation  |
                  | logicalop |

                  | = |
                  | [ |
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.

#### 2.3.1 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 algebraic quotient.

# 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* * floor(*A*/*B*))
where floor (*x*) = the largest integer '> *x*.

#### 2.3.2 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*).

##### 2.3.2.1 Numeric Relations

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

##### 2.3.2.2 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 empty string is a substring of every string.

The relation ] is called *follows*. *A* ] *B* is true if and only if *A* follows *B* in the 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*))).

### 2.3.3 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.

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

See 2.2 for the definition of expratom. See section 1 for the definition of V.

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

$$S = S_1\ S_2\ ...\ S_n$$

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

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

```
                              | patcode |
patatom  ::= repcount         |         |
                              | strlit  |
```

See 2.2.5 for the definition of strlit.

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

See 2.2.3 for the definition of intlit.

```
                  | C |
                  | N |
                  | P |
patcode   ::=     | A |  ...
                  | L |
                  | U |
                  | E |
```

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, $\text{intlit}_1$.$\text{intlit}_2$, 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 .$\text{intlit}_2$ , the lower bound is taken to be zero. If the upper bound is omitted, so that the range has the form $\text{intlit}_1$. , the upper bound is taken to be indefinite; that is, the range is at least $\text{intlit}_1$ occurrences. Then patatom is satisfied by the concatenation of a number of $S_i$, each of which meets the specification of patatom, where the number must be within the expressed or implied bounds of the specified range, inclusive.

The dual operator '? is defined by:

$A$ '? $B$ = '($A$ ? $B$)

#### 2.3.4 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.

$A$ ! $B$ = ( 0 if both $A$ and $B$ have the value 0 )
( 1 otherwise )

$A$ & $B$ = ( 1 if both $A$ and $B$ have the value 1 )
( 0 otherwise )

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

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

#### 2.3.5 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.

### 2.4 Routines

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

```
routine      ::=  routinehead routinebody
```

See 2.4.1 for the definition of routinebody.

```
routinehead  ::=  routinename eol

routinename  ::=  name
```

See 2.2.1 for the definition of name.

#### 2.4.1 Routine Structure

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

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

line        ::=  | formalline |
                 | levelline  |

formalline  ::=  label formallist ls linebody

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

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

See 2.5.3 for the definition of comment.

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

See 2.2.1 for the definition of name. See section 1 for the definition of L.

```
label       ::=  | name   |
                 | intlit |
```

See 2.2.1 for the definition of name. See 2.2.3 for the definition of intlit.

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

See 2.5 for the definition of command.

```
ls          ::=  SP ...

li          ::=  . [ SP ] ...

cs          ::=  SP ...

eol         ::=  CR LF

eor         ::=  CR FF
```

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. A formallist may only be present on a line whose LEVEL is one, i.e., does not contain an li.

#### 2.4.2 Routine Execution

MUMPS routines are executed in a sequence of blocks. Each block is dynamically defined and is invoked by the instance of an argumentless DO command, a doargument, an exfunc, or an exvar. Each block consists of a set of lines that all have the same LEVEL; the block begins with the line reference implied by the DO, exfunc, or exvar and ends with an implicit or explicit QUIT command. If no label is specified in the doargument, exfunc, or exvar, the first line of the routinebody is used. The *execution level* is defined as the LEVEL of the line currently being executed. Lines which have a LEVEL greater than the current execution level are ignored, i.e., not executed. An implicit QUIT command is executed when a line with a LEVEL less than the current execution level or the eor is encountered, thus terminating this block (see 2.6.15 for a description of the actions of QUIT). The initial LEVEL for a process is one. The argumentless DO command increases the execution level by one. (See also the DO command and GOTO command).

Within a given routine or subroutine execution proceeds sequentially from line to line in top to bottom order, starting with the line specified by the invoked label or first line of the routine if no label is given. Within a line execution begins at the leftmost command and proceeds left to right from command to command. Routine flow commands DO, ELSE, FOR, GOTO, IF, QUIT, XECUTE, exfunc and exvar extrinsic functions and variables, provide exception to this execution flow. Within a command, all expratoms are evaluated in a left-to-right order with all expratoms that occur to the left of the expratom being evaluated, including the complete resolution of any indirection, prior to the evaluation of that expratom, except as explicitly noted elsewhere in this document. The expratom is formed by the longest sequence of characters that satisfies the definition of expratom.

It is an error to begin execution of any formalline unless that formalline has just been reached as a result of an exvar, an exfunc, or a DO command doargument that contains an actuallist.

### 2.5 General command Rules

Every command starts with a *command word* which dictates the syntax 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]
N[EW]
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 2.6.

```
                  | syntax of BREAK command   |
                  | syntax of CLOSE command   |
                  |             .             |
command  ::=      |             .             |
                  |             .             |
                  | syntax of XECUTE 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 $arg_2$ ... .

### 2.5.1 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 2.2.4.2 for the definition of tvexpr.

If the postcond is absent or the postcond is present and the value of the tvexpr is true, the command is executed. If the postcond is present and the value of the tvexpr is false, the 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. In such cases the arguments' expratoms that occur prior to the postcond are evaluated prior to the evaluation of the postcond.

### 2.5.2 Spaces in Commands

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

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

b. If a command instance contains at least one argument, the 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.

### 2.5.3 Comments

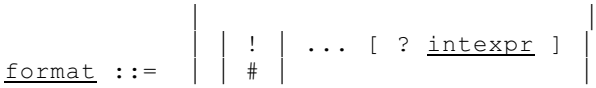
If a semicolon appears in the command word initial-letter position, it is the start of a comment. The remainder of the line to eol must consist of graphics only, but is otherwise ignored and nonfunctional.

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

See 2.1 for the definition of graphic.

### 2.5.4 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.



```
|          ? intexpr          |
```

See 2.2.4.1 for the definition of intexpr.

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.)

#### 2.5.5 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
```

#### 2.5.6 Timeout

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

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

See 2.2.4 for the definition of numexpr.

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

$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.

#### 2.5.7 Line References

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

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

See 2.4.1 for the definition of label. See 2.2 for the definition of expratom. See section 1 for the definition of V.

```
                    |    routinename        |
routineref  ::=     |                       |
                    | @ expratom V routineref |
```

See 2.4 for the definition of routinename. See 2.2 for the definition of expratom. See section 1 for the definition of V.

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

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

See 2.2.4.1 for the definition of intexpr.

If the delimiter ^ is absent, the routine being executed is implied. If the line reference (dlabel [+intexpr]) is absent, the first line is implied.

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

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

See 2.4.1 for the definition of label. See 2.4 for the definition of routinename.

If + intexpr is absent, the line denoted by dlabel is the one containing label in a defining occurrence. If + intexpr is present and has the value *n* '< 0, the line denoted is the *n*th line after the one containing label in a defining occurrence. A negative value of intexpr 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, GOTO, and JOB commands, as well as the $TEXT function, can refer to a line in a routine other than that in which they occur; this requires a means of specifying a routinename.

#### 2.5.8 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 SP L argument

the argument syntax will be expressed as

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

See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L. See section 1 for the definition of L.

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.

#### 2.5.9 Parameter Passing

Parameter passing is a method of passing information in a controlled manner to and from a subroutine as the result of an exfunc, an exvar, or a DO command with an actuallist.

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

See section 1 for the definition of L.

```
                        | . actualname  |
actual      ::=         |               |
                        | expr          |
```

See 2.3 for the definition of expr.

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

See 2.2.1 for the definition of name. See 2.2 for the definition of expratom. See section 1 for the definition of V.

When parameter passing occurs, the formalline designated by the labelref must contain a formallist in which the number of names is greater than or equal to the number of actuals in the actuallist. The correspondence between actual and formallist name is defined such that the first actual in the actuallist corresponds to the first name in the formallist, the second actual corresponds to the second formallist name, etc. Similarly, the correspondence between the parameter list entries, as defined below, and the actual or formallist names is also by position in left-to-right order. If the syntax of actual is .actualname, then it is said that the actual is of call-by-reference format; otherwise, it is said that the actual is of the call-by-value format.

When parameter passing occurs, the following steps are executed:

a. Process the actuals in left-to-right order to obtain a list of DATA-CELL pointers called the parameter list. The parameter list contains one item per actual. The parameter list is created according to the following rules:

1. If the actual is call-by-value, then evaluate the expr and create a DATA-CELL with a zero tuple value equal to the result of the evaluation. The pointer to this DATA-CELL is the parameter list item.

2. If the actual is call-by-reference, search the NAME-TABLE for an entry containing the actuallist name. If an entry is found, the parameter list item is the DATA-CELL pointer in this NAME-TABLE entry. If the actuallist name is not found, create a NAME-TABLE entry containing the name and a pointer to a new (empty) DATA-CELL. This pointer is the parameter list item.

b. Create the parameter frame on the PROCESS-STACK containing the formallist.

c. For each name in the formallist, search the NAME-TABLE for an entry containing the name and if the entry exists, copy the NAME-TABLE entry into the parameter frame and delete it from the NAME-TABLE. This step performs an implicit NEW on the formallist names.

d. For each item in the parameter list, create a NAME-TABLE entry containing the corresponding formallist name and the parameter list item (DATA-CELL pointer). This step *binds* the formallist names to their respective actuals.

As a result of these steps, two (or more) NAME-TABLE entries may point to the same DATA-CELL. As long as this common linkage is in effect, a SET or KILL of an lvn with one of the names appears to perform an implicit SET or KILL of an lvn with the other name(s). Note that a KILL does not undo this linkage of multiple names to the same DATA-CELL, although subsequent parameter passing or NEW commands may.

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

At the time of the QUIT, the formallist names are unbound and the original variable environment is restored. See 2.6.15 for a discussion of the semantics of the QUIT operation.

### 2.6 Command Definitions

The specifications of all commands follow.

#### 2.6.1 BREAK

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

See 2.5.1 for the definition of postcond.

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.

#### 2.6.2 CLOSE

```
C[LOSE] postcond SP L closeargument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.3 for the definition of expr. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

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

See 2.3 for the definition of expr.

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.

**2.6.3 DO**

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

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.5.7 for the definition of entryref. See 2.5.1 for the definition of postcond. See 2.5.7 for the definition of labelref. See 2.5.9 for the definition of actuallist. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

An argumentless DO initiates execution of an inner block of lines. If postcond is present and its tvexpr is false, the execution of the command is complete. If postcond is absent, or the postcond is present and its tvexpr is true, the DO places a DO frame containing the current execution location, the current execution level, and the current value of $T on the PROCESS-STACK, increases the execution level by one, and continues execution at the next line in the routine. (See 2.4.2 for an explanation of routine execution.) When encountering an implicit or explicit QUIT not within the scope of a subsequently executed doargument, xargument, exfunc, exvar, or FOR, execution of this block is terminated (see 2.6.15 for a description of the actions of QUIT). Execution resumes at the command (if any) following the argumentless DO.

DO with arguments is a generalized call to the subroutine specified by the entryref, or labelref, in each doargument. The line specified by the entryref or labelref, must have a LEVEL of one. Execution of a doargument to a line whose LEVEL is not one is erroneous.

If the actuallist is present in an executed doargument, parameter passing occurs and the formalline designated by labelref must contain a formallist in which the number of names is greater than or equal to the number of actuals in the actuallist.

Each doargument is executed, one at a time in left-to-right order, in the following steps.

a. Evaluate the expratoms of the doargument.

b. If postcond is present and its tvexpr is false, execution of the doargument is complete. If postcond is absent, or postcond is present and its tvexpr is true, proceed to the step c.

c. A DO-frame containing the current execution location and the execution level are placed on the PROCESS-STACK.

d. If the actuallist is present, execute the sequence of steps described in 2.5.9 Parameter Passing.

e. Continue execution at the first command following the ls of the line specified by entryref or labelref. Execution of the subroutine continues until an eor or a QUIT is executed that is not within the scope of a subsequently executed FOR, argumentless DO, doargument, xargument, exfunc, or exvar. The scope of this doargument is said to extend to the execution of that QUIT or eor. (See 2.6.15 for a description of the actions of QUIT.) Execution then returns to the first character position following the doargument.

**2.6.4 ELSE**

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

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.

### 2.6.5 FOR

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

See 2.2.2.1 for the definition of lvn. See section 1 for the definition of L.

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

See 2.3 for the definition of expr. See 2.2.4 for the definition of numexpr.

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.

The FOR with arguments specifies repeated execution of the commands within its scope for different values of the local variable lvn, under successive control of the forparameters, from left to right. Any expressions occurring in lvn, such as might occur in subscripts or indirection, are evaluated once per execution of the FOR, prior to the first execution of any forparameter.

For each forparameter, control of the execution of the commands in the scope is specified as follows. (Note that *A*, *B*, and *C* are hidden temporaries.)

a. If the forparameter is of the form $\underline{\text{expr}}_1$.

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

b. If the forparameter is of the form $\underline{\text{numexpr}}_1$ : $\underline{\text{numexpr}}_2$ : $\underline{\text{numexpr}}_3$ and $\underline{\text{numexpr}}_2$ is nonnegative.

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

1. Set *A* = $\underline{\text{numexpr}}_1$.

2. Set *B* = numexpr$_2$.
3. Set *C* = numexpr$_3$.
4. Set lvn = *A*.
5. If lvn < *C*, processing of this forparameter is complete.
6. Execute the commands in the scope once; 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 commands in the scope once; an undefined value for lvn is erroneous.
5. Set lvn = lvn + *B*.
6. Go to 4.

If the FOR command has no argument.

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

b. Goto 1.

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

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

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

a. It terminates that particular execution of the scope at the QUIT; commands to the right of the QUIT are not executed.

b. It causes any remaining values of the forparameter in control at the time of execution of the QUIT, and the remainder of the forparameters in the same forparameter list, not to be calculated and the commands in the scope not to be executed under their control.

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

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

**2.6.6 GOTO**

```
G[OTO] postcond SP L gotoargument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.5.1 for the definition of postcond. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

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

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

See 2.6.5 for a discussion of additional effects of GOTO when executed within the scope of FOR.

#### 2.6.7 HALT

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

See 2.5.1 for the definition of postcond.

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

#### 2.6.8 HANG

```
H[ANG] postcond SP L hangargument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

```
                    | numexpr                     |
hangargument  ::=   |                             |
                    | @ expratom V L hangargument |
```

See 2.2.4 for the definition of numexpr. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

Let *t* be the value of numexpr. If *t* '> 0, HANG has no effect. Otherwise, execution is suspended for *t* seconds.

#### 2.6.9 IF

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

See section 1 for the definition of L.

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

See 2.2.4.2 for the definition of tvexpr. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

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

#### 2.6.10 JOB

```
J[OB] postcond SP L jobargument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.5.7 for the definition of entryref and labelref. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

```
jobactuallist   ::=  ( L expr )
```

See 2.3 for the definition of expr. See section 1 for the definition of L.

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

See 2.5.6 for the definition of timeout.

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

See 2.3 for the definition of expr.

For each jobargument, the JOB command attempts to initiate another MUMPS process. If the jobactuallist is present in a jobargument, the formalline designated by labelref must contain a formallist in which the number of names is greater than or equal to the number of exprs in the jobactuallist.

The JOB command initiates this MUMPS process at the line specified by the entryref or labelref. If the jobactuallist is present, the process will have certain variables initially defined. These variables will be taken from the formallist of the formalline designated by the labelref. Formallist names will be created and paired with the values of the jobactuallist exprs for as many exprs as are present in the jobactuallist. There is no linkage between the started process and the process that initiated it; jobactuallist exprs are passed only by value. If the jobactuallist is not present, the process will have no variables initially defined.

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.

#### 2.6.11 KILL

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

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.2.2.2 for the definition of glvn. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

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

See 2.2.1 for the definition of name. See 2.2 for the definition of expratom. See section 1 for the definition of V.

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

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

See section 1 for the definition of L.

KILL is defined using a subsidiary function K(*V*) where *V* is a glvn.

1. Search for the name *V* in the NAME-TABLE. If no such entry is found, the function is completed. Otherwise, extract the DATA-CELL pointer and proceed to step 2.

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

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

Note that as a result of procedure *K*, $D(*V*)=0, i.e., the value of *V* is undefined, and *V* has no descendants.

The actions of the three forms of KILL are then defined as:

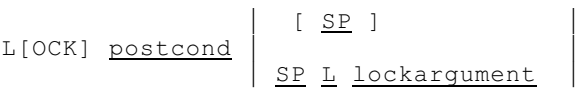
a. Selective Kill - apply *K* to the specified glvn.

b. Exclusive Kill - apply *K* to all names in the NAME-TABLE except those in the argument list. Note that the names in the argument list of an exclusive kill may not be subscripted.

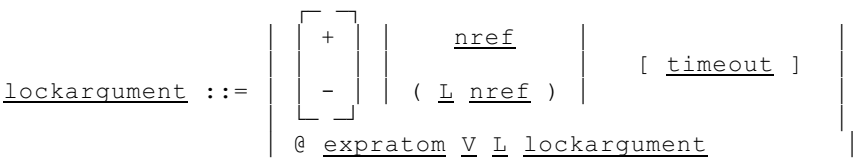
c. Kill All - apply *K* to all names in the NAME-TABLE.

If a variable *N*, a descendant of *M*, is killed, the killing of *N* affects the value of $D(*M*) as follows: if *N* was not the only descendant of *M*, $D(*M*) is unchanged; otherwise, if *M* has a defined value $D(*M*) is changed from 11 to 1; if *M* does not have a defined value $D(*M*) is changed from 10 to 0.

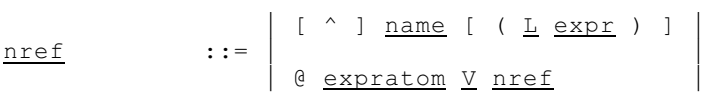
### 2.6.12 LOCK




See 2.5.1 for the definition of postcond. See section 1 for the definition of L.




See 2.6.12 for the definition of nref. See 2.5.6 for the definition of timeout. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.



See 2.2.1 for the definition of name. See 2.3 for the definition of expr. See 2.2 for the definition of expratom. See 2.6.12 for the definition of nref. See section 1 for the definition of V. See section 1 for the definition of L.

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 of the naked indicator. Its use is not required to access globals, nor does its use inhibit other processes from accessing globals. It is an interlocking mechanism whose use depends on programmers establishing and following conventions.

Each lockargument specifies a subspace of the total MUMPS name space for which the executing process seeks to make or release an exclusive claim; the details of this subspace specification are given below.

For the purposes of this discussion, *name space* is herein defined as the union of all possible nrefs after resolution of all indirection. There exists a table, called the locktable, which contains zero or more nrefs for each MUMPS process. A given nref may appear more than once for the same process and it may not appear for multiple processes. Each nref represents a claim on a portion of the name space. The subspace of the total name space claimed by each nref in the locktable is 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, let the nref be $N(s_1,s_2,...,s_n)$. Then the subspace is the set of the following points: one point for each of $N$, $N(s_1)$, $N(s_1,s_2)$,...,$N(s_1,...,s_i)$, where $i > n$, and one point for each descendant (see 2.2.7 $DATA function for a definition of descendant) of nref.

If the LOCK command is argumentless, LOCK removes all nrefs in the locktable that are associated with this process.

Execution of lockargument occurs in the following order:

a. Any expression evaluation involved in processing the lockargument is performed.

b. When the form of lockargument does not include an initial + or - sign, then prior to evaluating or executing the rest of the lockargument, LOCK first removes all nrefs in the locktable that are associated with this process. For the rest of the discussion, this form acts the same as if an initial + sign were present.

c. If an explicit or implicit leading + sign is present, then:

1. A test is made to see if this process can claim the entire subspace defined by the lockargument. This subspace can be claimed if each nref of the lockargument does not intersect the union of the subspaces claimed at this instant by nrefs in the locktable for all other processes.

2. If the test performed above indicates that the process cannot claim the specified subspace, execution of this process is suspended until repetition of the test would indicate that the process can claim the specified subspace, or, when a timeout is present, until the timeout expires, if that occurs first. If the timeout expires, step 3 below is skipped.

3. All of the nrefs in the lockargument are inserted into the locktable for this process. This may result in some nrefs being in the table more than once for this process. The nrefs for the lockargument are either inserted all at once or not at all.

d. If the lockargument has a leading - sign, then for each nref in the lockargument, if the nref exists in the locktable for this process, one instance of nref is removed from the locktable.

e. If a timeout is present, the condition reported by $T upon completion of the execution of the lockargument is the success or failure to establish or relinquish the claim; it has the value of 1 if the lock claim is established or 0 if the timeout expires. If no timeout is present, execution of the lockargument does not change $T.

**2.6.13 NEW**

```
                    | [ SP ]                |
N[EW] postcond      |                       |
                    | SP L newargument      |
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

```
                    | lname                          |
                    |                                |
newargument ::=     | ( L lname )                    |
                    |                                |
                    | @ expratom V L newargument     |
```

See 2.6.11 for the definition of lname. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

NEW provides a means of performing variable scoping.

The three argument forms of NEW are given the following names:

a. lname: Selective New
b. (L lname): Exclusive New
c. Empty argument list: New All

See section 1 for the definition of L.

Each argument of the NEW command pushes a frame containing the NEW argument onto the PROCESS-STACK and copies a set of NAME-TABLE entries into the frame.

The actions of the three forms of NEW are then defined as:

a. Selective New - the NAME-TABLE entry for lname is copied into the frame.

b. Exclusive New - the set of NAME-TABLE entries for all names except the names in the argument are copied into the frame.

c. New All - all entries in the NAME-TABLE are copied into the frame.

In all three cases, the NAME-TABLE entries copied into the frame are subsequently deleted from the NAME-TABLE. This deletion has the effect of making the variable unknown in the current process context.

See 2.6.15 for a description of the actions of QUIT.

**2.6.14 OPEN**

```
O[PEN] postcond SP L openargument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.3 for the definition of expr. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

```
                          | deviceparameters [ timeout ] |
openparameters  ::=       |                              |
                          | timeout                      |
```

See 2.6.2 for the definition of deviceparameters. See 2.5.6 for the definition of 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 2.6.2 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 2.6.18)

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.

#### 2.6.15 QUIT

```
                        | [ SP ]   |
Q[UIT] postcond         |          |
                        | SP expr  |
```

See 2.5.1 for the definition of postcond. See 2.3 for the definition of expr.

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

Encountering the end-of-routine mark eor is equivalent to an unconditional argumentless QUIT.

The effect of executing QUIT in the scope of FOR is fully discussed in 2.6.5. 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 argumentless DO command, doargument, xargument, exfunc, or exvar if not explicitly then implicitly, because the initial activation of a process, including that due to execution of a jobargument, may be thought of as arising from execution of

a DO naming the first executed routine of that process.

The effect of executing a QUIT in the scope of an argumentless DO command, doargument, xargument, exfunc, or exvar is to restore the previous variable environment (if necessary), restore the value of $T (if necessary), restore the previous execution level, and continue execution at the location of the invoking argumentless DO command, doargument, xargument, exfunc, or exvar.

If the expr is present in the QUIT, this return must be to an exfunc or exvar. Similarly, if the expr is not present, the return must be to an argumentless DO command, doargument or xargument. Any other case is erroneous.

The following steps are executed when a QUIT is encountered:

a. If an expr is present, evaluate it. This value becomes the value of the invoking exfunc or exvar.

b. Remove the frame on the top of the PROCESS-STACK. If no such frame exists, then execute an implicit HALT.

c. If that frame is from a NEW, examine the saved argument of the NEW and take one of the following actions dependent on the argument types:

1. Selective New - perform an implicit KILL on the argument lname.

2. Exclusive New - perform an implicit KILL on all names in the NAME-TABLE except those in the argument of the NEW.

3. New All - perform an implicit KILL ALL.

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

Processing of this frame is complete, continue at step b.

d. If the frame is a parameter frame, extract the formallist and process each name in the list with the following steps:

1. Search the NAME-TABLE for an entry containing the name. If no such entry is found, processing of this name is complete. Otherwise, proceed to step 2.

2. Delete the NAME-TABLE entry for this name.

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

Processing of this frame is complete, continue at step b.

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

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

#### 2.6.16 READ

```
R[EAD] postcond SP L readargument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.2.5 for the definition of strlit. See 2.5.4 for the definition of format. See 2.2.2.1 for the definition of lvn. See 2.5.6 for the definition of timeout. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

```
readcount     ::=  # intexpr
```

See 2.2.4.1 for the definition of 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 (see 2.2.2.3 for a description of the value assignment operation). If no timeout is present, execution will be suspended until the input message is terminated, either explicitly or implicitly with a readcount. (See 2.6.18 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 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* '> 0. Otherwise, the input message is a string whose length is at most *n* characters, and which is terminated by an implementor-defined, possibly device-dependent procedure, which may be the receipt of the *n*th character. If timeout is present and the timeout expires prior to the termination of the input message by either mechanism just described, the value given to 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 2.5.4 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 writeargument equal, in turn, to the final value of the respective lvn resulting from the READ.

### 2.6.17 SET

```
S[ET] postcond SP L setargument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.2.2.2 for the definition of glvn. See 2.3 for the definition of expr. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

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

See 2.2.2.2 for the definition of glvn. See 2.3 for the definition of expr. See 2.2.4.1 for the definition of intexpr.

SET is the general means both for explicitly assigning values to variables, and for substituting new values in pieces of a variable. Each setargument computes one value, defined by its expr. That value is then either assigned to each of one or more variables, or it is substituted for one or more pieces of a variable's current value. Each variable is named by one glvn.

Each setargument is executed one at a time in left-to-right order. The execution of a setargument occurs in the following order.

a. One of the following two operations is performed:

   1. If the portion of the setargument to the left of the = consists of one or more glvns, the glvns are scanned in left-to-right order and all subscripts are evaluated, in left-to-right order within each glvn.

   2. If the portion of the setargument to the left of the = consists of a setpiece, the glvn that is the first argument of the setpiece is scanned in left-to-right order and all subscripts are evaluated in left-to-right order within the glvn, and then the remaining arguments of the setpiece are evaluated in left-to-right order.

b. The expr to the right of the = is evaluated.

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. (See 2.2.2.3 for a description of the value assignment operation).

   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 *m*th through the *n*th pieces of the current value of the glvn, where the value of

*d* is the piece delimiter. Note that both *m* and *n* are optional. If neither is present, then *m* = *n* = 1; if only *m* is present, then *n* = *m*. If glvn has no current value, the empty string is used as its current value. Note that the current value of glvn is obtained just prior to replacing it. That is, the other arguments of setpiece are evaluated in left-to-right order, and the expr to the right of the = is evaluated prior to obtaining the value of glvn.

Let *s* be the current value of glvn, *k* be the number of occurrences of *d* in *s*, that is, *k* = max(0,$L(*s*,*d*) - 1), and *t* be the value of expr. The following cases are defined, using the concatenation operator _ of 2.3.5.

a) *m* > *n* or *n* < 1. The glvn is not changed and does not change the naked indicator.

b) *n* '< *m*-1 > *k*. The value in glvn is replaced by *s*_F(*m*-1-*k*)_*t*, where F(*x*) denotes a string of *x* occurrences of *d*, when *x* > 0; otherwise, F(*x*) = "". In either case, glvn affects the naked indicator.

c) *m*-1 '> *k* < *n*. The value in glvn is replaced by $P(*s*,*d*,1,*m*-1)_F(min(*m*-1,1))_*t*.

d) Otherwise, The value in glvn is replaced by $P(*s*,*d*,*m*-1)_F(min(*m*-1,1))_*t*_*d*_$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.

### 2.6.18 USE

```
U[SE] postcond SP L usearqument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.3 for the definition of expr. See 2.6.2 for the definition of deviceparameters. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

The value of the first expr of each usearqument 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 2.6.2 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 usearqument, 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.

#### 2.6.19 VIEW

```
V[IEW] postcond arguments unspecified
```

See 2.5.1 for the definition of postcond.

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.

#### 2.6.20 WRITE

```
W[RITE] postcond SP L writeargument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

```
                        | format                          |
writeargument  ::=      | expr                            |
                        | * intexpr                       |
                        | @ expratom V L writeargument    |
```

See 2.5.4 for the definition of format. See 2.3 for the definition of expr. See 2.2.4.1 for the definition of intexpr. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

The writeargument**s** 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 2.5.4 are executed. Each character of output, in turn, affects $X and $Y as described in 2.5.4 and 2.5.5.

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 2.5.5.

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.

#### 2.6.21 XECUTE

```
X[ECUTE] postcond SP L xargument
```

See 2.5.1 for the definition of postcond. See section 1 for the definition of L.

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

See 2.3 for the definition of expr. See 2.5.1 for the definition of postcond. See 2.2 for the definition of expratom. See section 1 for the definition of V. See section 1 for the definition of L.

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

Each xargument is evaluated one at a time in left-to-right order. If the postcond in the xargument is present and its tvexpr is false, the xargument is not executed. Otherwise, if the value of expr is *x*, execution of the xargument is executed in a manner equivalent to execution of DO *y*, where *y* is the spelling of an otherwise unused label attached to the following two-line subroutine considered to be a part of the currently executing routine.

```
y  ls   x    eol

   ls  QUIT  eol
```

See 2.4.1 for the definition of ls and eol.

**2.6.22 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.

# Part 2: MUMPS Portability Requirements

## Table of Contents

# American National Standard for Information Systems - Programming Languages - MUMPS (Part 2: MUMPS Portability Requirements)

## Introduction

Part 2 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. 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.

# 1 Expression Elements

## 1.1 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.

## 1.2 Local Variables

### 1.2.1 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 5).

### 1.2.2 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 two times the number of subscripts, plus the length of the local variable name must not exceed 127.

### 1.2.3 Values of Subscripts

Local variable subscript values are nonempty strings which may only contain characters from the ASCII printable character subset. The length of each subscript is limited to 63 characters. When the subscript value satisfies the definition of a numeric data value (See 2.2.3.1 of the MUMPS Language Specification), it is further subject to the restrictions of number range given in 1.5. 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.

### 1.2.4 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 (see 1.2.1) and the limit on the number of subscripts (see 1.2.2) place implicit limits on the number of distinct nodes which may be defined.

## 1.3 Global Variables

### 1.3.1 Number of Global Variables

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

### 1.3.2 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 two times the number of subscripts, plus the length of the global variable name must not exceed 127. 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.

#### 1.3.3 Values of Subscripts

The restrictions imposed on the values of global variable subscripts are identical to those imposed on local variable subscripts (see 1.2.3).

#### 1.3.4 Number of Nodes

There is no limit on the distinct global variable nodes which are defined.

### 1.4 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.

### 1.5 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 precision of any value used in arithmetic operations requiring a numeric interpretation is twelve significant digits.

Programmers should exercise caution in the use of noninteger arithmetic. In general, arithmetic operations on noninteger operands or arithmetic operations which produce noninteger results cannot be expected to be exact. In particular, noninteger arithmetic can yield unexpected results when used in loop control or arithmetic tests.

### 1.6 Integers

The magnitude of the value resulting from an integer interpretation is limited by the accuracy of numeric values (see 1.5). The values produced by integer valued operators and functions also fall within this range (see 2.2.4.1 of the MUMPS Language Specification for a precise definition of integer interpretation).

### 1.7 Character Strings

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

### 1.8 Special Variables

The special variables $X and $Y are nonnegative integers (see 1.6). 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 2.5.5 of the MUMPS Language Specification)

## 2 Expressions

### 2.1 Nesting of Expressions

The number of levels of nesting in expressions is not explicitly limited. The maximum string length does impose an implicit limit on this number (see 1.7).

### 2.2 Results

Any result, whether intermediate or final, which does not satisfy the constraints on character strings (see 1.7) is erroneous. Furthermore, integer results are erroneous if they do not satisfy the constraints on integers (see 1.6).

## 3 Routines and Command Lines

### 3.1 Command Lines

A command line (line) must satisfy the constraints on character strings (see 1.7). The length of a command line is the number of characters in the line up to but not including the eol.

The characters within a command line are restricted to the 95 ASCII printable characters. The character set restriction places a corresponding implicit restriction upon the value of the argument of the indirection delimiter (Section 4).

### 3.2 Number of Command Lines

There is no explicit limit on the number of command lines in a routine, subject to storage space restrictions (Section 5).

### 3.3 Number of Commands

The number of commands per line is limited only by the restriction on the maximum command line length (see 3.1).

### 3.4 Labels

A label of the form name is subject to the constraints on names; labels of the form intlit are subject to the length constraint on names (see 1.1).

### 3.5 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.

### 3.6 Number of Routines

There is no explicit limit on the number of routines. The number of routines is implicitly limited by the name length restriction (see 1.1).

## 4 Indirection

The values of the argument of indirection and the argument of the XECUTE command are subject to the constraints on character string length (see 1.7). They are additionally restricted to the character set limitations of command lines (see 3.1).

## 5 Storage Space Restrictions

The size of a single routine must not exceed 5000 characters. The size of a routine is the sum of the sizes of all the lines in the routine. The size of each line is its length (as defined in 3.1) plus two.

The size of local variable storage must not exceed 5000 characters. This size is defined as the sum of the sizes of all defined local variables, whether within the current NEW context or defined in a higher level NEW context. The size of an unsubscripted local variable is the length of its name in characters plus the length of its value in characters, plus four. The size of a local array is the sum of the following:

a) The length of the name of the array.

b) Four characters plus the length of each value.

c) The size of each subscript in each subscript list.

d) Two additional characters for each node *N*, whenever $DATA(*N*) is 10 or 11.

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

## 6 Nesting

Each active DO, Extrinsic Function, Extrinsic Special Variable, FOR, XECUTE, and indirection occurrence is counted as a level of nesting. Control storage provides for thirty levels of nesting. The actual use of all these levels may be limited by storage restrictions (Section 5).

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 5).

## 7 Other Portability Requirements

Programmers should exercise caution in the use of noninteger values for the HANG command and in timeouts. In general, the period of actual time which elapses upon the execution of a HANG command cannot be expected to be exact. In particular, relying upon noninteger values in these situations can lead to unexpected results.