MUMPS
Development
Committee

MUMPS

MUMPS

PROGRAMMERS' REFERENCE MANUAL

MUMPS DEVELOPMENT COMMITTEE

MUMPS

Programmers' Reference Manual

MUMPS Development Committee
Subcommittee on Documentation
Task Group #1 on the Reference Manual

This reference manual is based on the language defined in the
MUMPS Language Specification, MDC/28, dated March 12, 1975, with
corrigenda through March 9, 1976 (also published as Part I of National
Bureau of Standards Handbook 118, dated January and March 9, 1976,
respectively). It supersedes the June 20, 1975 manual having the
same title and document number.

The reader is hereby notified that the following specification
has been approved by the MUMPS Development Committee but that it may
be a partial specification which relies on information appearing in
many parts of the MUMPS specifications. The specification is dynamic
in nature, and the changes reflected by this approved release may not
correspond with the latest specification available.

Because of the evolutionary nature of MUMPS specifications, the reader
is further reminded that changes are likely to occur in the specification
released herein prior to a complete republication of MUMPS speci-
fications.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

*Note: Except for Chapters 1, 3, and 7, each chapter starts with a
       Table of Contents at the page indicated.

CHAPTER 1

PURPOSE OF THIS MANUAL

This document is an application programmers' reference manual for the use of the Standard MUMPS Language. Since it is written independently of any implementation of the MUMPS language, and since many implementations of MUMPS will be accompanied by their own application programmers' manuals, the question arises: What is the role of an implementation-independent reference manual? The significance of this question is heightened by the lack of a need for a teaching manual for the Standard MUMPS Language; the MUMPS Primer fills that role.

This reference manual addresses itself primarily to the programmer who already knows the Standard MUMPS Language well enough to write application programs. It serves such a programmer by means of the following approach.

1. The Reference Manual is for reference use. Once the use of the manual is understood (see Chapter 2), it is used to give specific answers to specific questions. After reading Chapter 2 and obtaining a familiarity with the rest of the document, the user should be able to dip into any portion of the manual and stay only until his specific question is answered.

2. The Reference Manual concentrates on questions of interpretation, rather than language structure. Syntax is presented, but the primary emphasis is on answering the "What happens when...?" type of question. As a result of this emphasis, much of the explanation of this manual is presented in terms of concrete actions of a "MUMPS System Model", described initially in Chapter 3. This MUMPS System Model is a hypothetical implementation of Standard MUMPS invented only for the expository purposes of this manual. Real implementations are not expected necessarily to imitate the internal characteristics of the System Model; however, if the MUMPS routines in question observe all MUMPS portability rules, a valid implementation of Standard MUMPS and the MUMPS System Model should give identical output, given identical input.

3. The Reference Manual attempts to illuminate the remote corners of the language as well as the most frequently used and most easily understood portions.

4. The Reference Manual encourages program portability by incorporating the MUMPS Portability Requirements. These requirements are not merely incorporated, however, but are explicitly cited. One important reason for visibly identifying the Portability Requirements is to avoid the appearance of contradicting the documentation of those implementations which relax some of these restrictions. By these means the responsibility for trading off portability against other benefits possibly offered by some implementations is placed where it belongs: with the management of the programming activity.

Secondarily, this Reference Manual addresses itself to the implementor. Implementors are constantly having to interpret language specifications by answering questions which were not anticipated by the language designers. This manual attempts to anticipate as many questions as possible. To cover the inevitable unanticipated questions, the use of a comprehensive model seeks to present a coherent structure within which different implementors will tend to make similar interpretations. This goal is particularly important to the designers of MUMPS, who have paid great attention to questions of application portability.

CHAPTER 2

A DESCRIPTION OF THE STYLE OF THIS MANUAL

Table of Contents

CHAPTER 2

A DESCRIPTION OF THE STYLE OF THIS MANUAL

2.1 Divisions of this Manual

As the Table of Contents indicates, this Reference Manual is divided into three parts.

Part I (Chapters 1-2) contains introductory material.

Part II (Chapters 3-6) presents a model of a MUMPS system. In Part II various concepts are introduced which are later employed in the language descriptions. Thus many of the cross-references in Part III refer back to Part II.

Part III (Chapters 7-12) describes the syntax and semantics of the elements of the MUMPS language, generally in top-down order, from routines (Chapter 7) to expression atoms (Chapter 12).

In addition, there are three appendices.

Appendix A contains a table of ASCII characters whose particular value to the MUMPS programmer is that the numeric character codes are presented in decimal form.

Appendices B and C contain indices to the definitions of the technical terms appearing in this manual.

2.2 Technical Terminology

Technical terms introduced in this manual fall mainly into two categories.

Syntactic Types

A syntactic type is a name given to a class of character strings which may appear in a MUMPS system. Membership in the class is defined by the spelling of the strings.

It is important to note that the strings classified by syntactic types can occur in several different contexts.

1. They may occur as part of a MUMPS routine. Examples are line, command, expression, string literal.

2. They may occur in data operated on by a MUMPS process. Examples are numeric value, nonnegative integer, truth value.

3. They may occur as values computed by a process defined in the model, but not necessarily appearing as data. Examples are node designator, device designator.

Thus, the syntax definition and specification methods employed in this manual apply uniformly to data strings as well as routine strings.

Syntactic types are always spelled entirely with lower-case letters (and an occasional hyphen). In certain contexts they may be underlined; this underlining does not change their nature, but is meant only to assist in rendering the accompanying description more readable.

1. When a syntactic type occurs in a syntax definition, it is always underlined. This underlining is particularly important when the type name has more than one word. Thus, expression atom is clearly a single entity.

2. In text, underlining a syntactic type is sometimes used to clarify a reference to a particular instance of that syntactic type. For example, in the syntax description of the SET command (8.2.15), an expression appears to the right of the = sign. Well down into the verbal description of the execution of the command the sentence appears: "The expression is evaluated". The use of underlining particularizes the reference so that it is clearly to the expression appearing in the syntax description.

Capitalized Technical Terms

The MUMPS System Model description introduces some technical terms, such as names of storage registers in the model, which are called here Capitalized Technical Terms. They are distinguished from syntactic types by appearing with initial capital letters in their spellings. Examples are Named System Storage, Job Number P-vector, Clock Register, Test Switch, Current Device Designator, Naked Indicator.

2.3 Syntax Specification Metalanguage

A syntax specification is a rule which partitions the set of all strings into two classes: those which satisfy the syntax specification and those which do not. Syntax specifications occur in two contexts.

1. On the right side of a syntax definition. A syntax definition has three parts: on the left is the name of the syntactic type being defined, in the middle is the metalangauge operator ::= , and on the right is a syntax specification. Examples from Chapter 7 follow.

$$\underline{line} \quad ::= \quad \underline{line\ head}\ \underline{line\ body}$$

$$\underline{line\ head} \quad ::= \quad [\underline{label}]\ \underline{ls}$$

$$\underline{line\ body} \quad ::= \quad \left[\begin{array}{c} \underline{command}\ [\ \sqcup\ \underline{command}]\ \ldots\ [\ \sqcup\ \underline{comment}] \\ \underline{comment} \end{array} \right] \underline{eol}$$

In the first example, the syntax type being defined is <u>line</u>, and the ::= says that, by definition, any (and only any) string which satisfies the syntax specification <u>line head</u> <u>line body</u> will satisfy the syntactic type <u>line</u> . Each syntactic type appears on the left side of at most one syntax definition (which may, however, be repeated within the manual). Those syntactic types which do not appear on the left side of a syntax definition, such as <u>alpha</u>, <u>graphic</u>, <u>nonquote</u> are defined informally; in the cases shown here (but not necessarily always) they are defined as a class of one-character strings.

2. As a free-standing syntax specification. For example, the definition of the syntactic type <u>command</u> is never formally given, but informally, <u>command</u> is defined to be satisfied by any of the syntax specifications of the separate commands described in 8.2. For example, the SET command has a (simplified) syntax specification as follows.

$$SET\ \sqcup\ \underline{storage\ reference}\ =\ \underline{expression}$$

The following metalinguistic operations occur in syntax specifications.

Succession

Example: <u>line head</u> <u>line body</u>

Explanation: Any string which can be expressed as a concatenation of

1. a string satisfying <u>line head</u> on the left, and
2. a string satisfying <u>line body</u> on the right,

satisfies <u>line head</u> <u>line body</u>.

Choice

Example:

| expression atom |
| expression expression tail |

Explanation: The vertical lines are used to define the boundaries of a string under consideration, and have no other metalinguistic significance. Sometimes (as in the definition of line body given before) this function of boundary definition is performed by brackets [ ], which have an additional metalinguistic function as explained below.

Any string which satisfies either the syntax specification expression atom or the syntax specification expression expression tail satisfies the syntax specification

| expression atom |
| expression expression tail | .

Option

Example: [label] ls

Explanation: Anything enclosed in brackets may be either present or absent. In the example, a string satisfies [label] ls either if it satisfies label ls or if it satisfies ls .

Repetition

Example: command | ⌣ command | ... eol

Explanation: The three dots ... denote optional indefinite repetition of the immediately preceding minimum syntax specification, in this case ⌣ command . Thus, a string satisfying any of the following satisfies the specification in the example.

command ⌣ command eol

command ⌣ command ⌣ command eol

command ⌣ command ⌣ command ⌣ command eol

etc.

Value Specification

Example: @ <u>expression atom</u> V <u>global variable</u>

Explanation: The metalinguistic operator <u>V</u> applies to the <u>expression atom</u> to its left and to the minimum syntax specification immediately to its right. It specifies that

1. the syntax of the syntax specification must satisfy
   @ <u>expression atom</u> , and
2. after the <u>expression atom</u> is evaluated, the syntax of its value must satisfy <u>global variable</u> .

Note: @ <u>expression atom</u> V is used to denote opportunities for indirection. Any discussion accompanying such a syntax specification applies to the string after all evaluation of indirection calls has occurred.

2.4   Starting Syntax Definitions

The following syntax definitions are taken as axiomatic.

<u>graphic</u>   ::=   the set of one-character strings whose characters are the 95 ASCII graphics

<u>alpha</u>   ::=   the set of one-character strings whose characters are the 52 ASCII alphabetics

⌴   ::=   the one-character string consisting of the ASCII character space (SP)

Any ASCII graphic may appear in a syntax specification; it stands for the one-character string consisting of that character. (This provides another reason that syntactic types are always underlined when appearing in syntax specifications.) The following nonprinting ASCII characters may also occur in syntax specifications: space (represented by SP or ⌴), carriage return (represented by CR), line feed (represented by LF), and form feed (represented by FF).

2.5   Portability Requirements

Statements taken from the MUMPS Portability Requirements are enclosed in the brackets

[Port: ...   :Port]   .

It is to be understood that such statements are not statements about the MUMPS language but are constraints which application programmers and MUMPS system implementors are asked to accept in order to make the goal of inter-system portability of application programs at reasonable cost a reality.

2.6   The Use of Examples

The text of this manual usually offers a precise basis from which the reader may make correct deductions. The examples, on the other hand, rely on the power of the reader to make correct inductions from statements which may not be formally complete or even correct. Thus the tone of the text accompanying the examples is more relaxed, and frequently there are inferences to be gained from the comparison of several examples.

Ocassionally, the word "typically" appears in the text of this manual. Statements containing "typically" are not to be taken as precise but as describing common usage.

CHAPTER 3

THE MUMPS SYSTEM MODEL

For the positive integer P, a P-partition MUMPS system is one capable of supporting the concurrent execution of at most P MUMPS processes. Each "partition" is an environment for the execution of one process. The number P is usually a fixed (or very slowly changing) attribute of a given system, but this is not necessarily so.

The model employed here of a P-partition MUMPS system consists of P+1 components: the P partitions plus System Storage.

Partitions

1	2	...	P
System Storage			
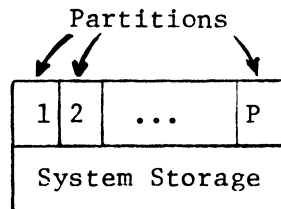
System Storage contains those data structures available to all partitions. The bulk of System Storage consists of what is traditionally called global data. In order to maintain a wide distinction between language entities and data, we shall use the term "Named System Storage" for that part of System Storage traditionally called global data.

In addition, System Storage contains a clock and three P-part data structures: the Lock List P-vector, the Open List P-vector, and the Job Number P-vector. Each partition has its own Lock List, Open List, and Job Number; the reason that they are assigned to System, and not Partition, Storage is that in manipulating the contents of its Lock and Open Lists each partition must know the contents of the P-1 other lists of the same type. Similarly, Job Numbers are assigned to be unique. The model is designed according to the premise that each partition may communicate with System Storage, but no two partitions may communicate directly with each other. The detailed structure of System Storage is described in Chapter 4.

Within each partition are the following components.
1.   An interpreter
2.   Partition Storage
3.   The Partition Stack

Each partition is given its own interpreter in this model, not as
an expression of a position on distributed vs. centralized processing,
but as a way of avoiding having to take positions on time-slicing methods.
As a further step in the direction of avoiding sequence dependencies arising
from system (as opposed to application) design decisions, the interactions
between the several interpreters and System Storage (reading values, writing
values, killing subtrees, creating chains, creating and deleting lists,
etc.) are defined to be indivisible and to occur nonsimultaneously. What
is not defined here is how fast each interpreter works, either absolutely
or in relation to any other interpreter.

Partition Storage contains those partition-local data structures
which are not stacked by the DO or XECUTE commands; these include Named
Partition Storage (i.e., local data or the "symbol table"), the Naked
Indicator, the Test Switch ($T), and the Current Device Designator. Par-
tition Storage is described in detail in Chapter 5.

The Partition Stack contains those entities which must be stacked
for proper sequence control. In this model the whole routine body is
stacked as well as status information such as the line pointer, line buffer,
and character pointer. The model is not concerned with the details of
stacking during expression evaluation, so there is no value stack as such
in the model. Chapter 6 describes in detail the Partition Stack and its
relationship to execution sequencing.

CHAPTER 4

SYSTEM STORAGE

Table of Contents

CHAPTER 4

SYSTEM STORAGE

System Storage in a P-partition system consists of the following
elements.
1. The Named System Storage data structure.
2. The following P-vectors in each of which there is a one-to-one
    correspondence between the partitions and the components of
    the P-vector.
        a. The Job Number P-vector. Each element is an integer.
        b. The Lock List P-vector. Each element is a list of
            Named System Storage node designators or Named Partition
            Storage node designators.
        c. The Open List P-vector. Each element is a list of
            Device Designators.
3. The Clock Register.

4.1 Named System Storage

    Named System Storage is a finite tree structure on every node
of which resides an "attribute block" storage register whose contents
are subject to certain constraints described below. The tree may
be thought of as being arranged in levels wherein all the nodes at
level n are at a distance n from the root ("directory") node.

Directory level (level 0)

Unsubscripted level (level 1)

Singly subscripted level (level 2)

Doubly subscripted level (level 3)

.
.
.

n-tuply subscripted level (level n+1)
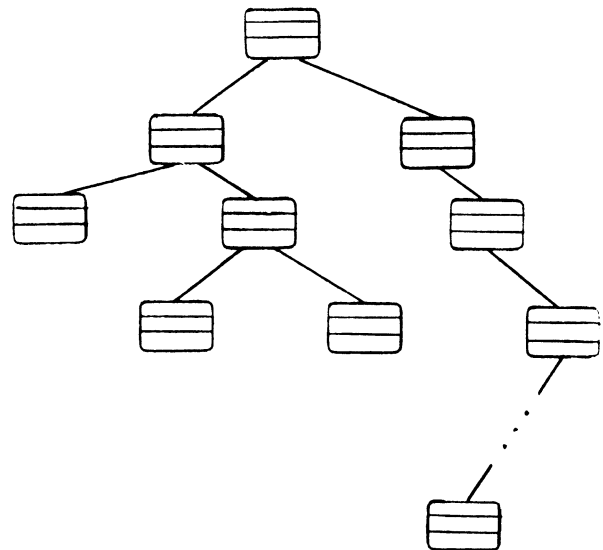
Each attribute block (shown as a rectangle above) is a three-part storage register containing the attributes named Name, D, Value.

Name:	^INV ¢ 3 ¢ 5
D:	1
Value:	2x4x8:160

Rules Governing Contents of Attribute Blocks At Each Node

1. The permissible entries in the D attribute are the four data strings

   0
   1
   10
   11

2. If D is 0 or 10, the Value attribute is said to be "undefined", and the content of the Value entry in the attribute block is unavailable. If D is 1 or 11, the Value attribute is said to be "defined", and the content of the Value entry in the attribute block may be any data string, including the empty string.

3. The Name attribute at level k (k≠0) is a k-tuple

   $$N \ ¢ \ S\underline{1} \ ¢ \ S\underline{2} \ ¢ \ ... \ ¢ \ S\underline{k-1} \qquad , \text{where}$$

   a. The separator ¢ is chosen for typing convenience and denotes a unique separator outside of the data character set.
   b. Each element of the k-tuple is a data string.
   c. [Port: The spelling of each S_i data string satisfies the syntax of <u>nonnegative integer</u> (see 12.4). :Port] (Note: See 2.5 for a description of this portability notation.)
   d. The spelling of the N data string satisfies the syntax of ^<u>name</u> (see 12.4).

Note that the nodes or attribute blocks are considered to be part of the data structure, but the branches are not; the branches are drawn only to show ascendancy relationships (see below).

Rules Governing Coupling Between Attribute Block Contents and Tree Structure

1. Every node has an attribute block.
2. The attribute block on the node at the directory level (the "directory node") has only one entry of interest, D, whose contents may be only 0 or 10.
3. The initial state of the tree (prior to the first process initiation) is as follows: there is only the directory node, for which D=0.
4. In the following definitions, let N(i) denote any node at level i.
    a. If N(k) and N(k+1) are connected by a branch, then N(k+1) is called an "immediate descendant" of N(k) and N(k) is called the "immediate ascendant" of N(k+1).
    b. N(j) is an "ascendant" of N(k) and N(k) is a "descendant" of N(j) (for j < k) if and only if there is a sequence of nodes N(j), N(j+1), ..., N(k) such that for each i in [j,k-1] N(i) is the immediate ascendant of N(i+1).
    Rule: For any positive j, N(j) is an immediate ascendant of N(j+1) if and only if the content of the Name attribute of N(j) may be obtained from the content of the name attribute of N(j+1) by deleting ¢ Sj .
5. If the Node N(k) has D=0 or 1, then N(k) has no descendant. The converse is not necessarily true; N(k) may be a "terminal node" (i.e., it has no descendant) with D=10 or 11. This situation can only arise, however, as the direct result of the execution of a selective KILL naming the sole immediate descendant of N(k).

4.2   Lock List P-vector

    Each partition of the system, numbered 1 to P, is associated with a "Lock List". The collection of these P Lock Lists makes up the "Lock List P-vector" in System Storage. Each Lock List may be empty, or it may be a list of data strings satisfying the syntax of node designator and separated by a unique separator outside of the data character set, designated here by ¢¢.

    node designator stands for the kinds of k-tuples (k≠0) which may occur in the Name part of the attribute block on a k-level node in either Named System Storage or Named Partition Storage:

        N ¢ S1 ¢ S2 ¢ ... ¢ Sk-1   , where

        a. The separator ¢ is outside of the ASCII data alphabet.
        b. Each entry of the k-tuple is a data string.
        c. [Port: The spelling of each Si data string satisfies the syntax of nonnegative integer (see 12.4). :Port]
        d. The spelling of the N data string satisfies the syntax of name or of ^name (see 12.4).

Lock List 1:   [<u>node designator</u> [¢¢ <u>node designator</u>] ...]

.
.
.

Lock List P: [<u>node designator</u> [¢¢ <u>node designator</u>] ...]

<u>Descendant Exclusivity Rule</u>
No node designator in Lock List i may be an initial subsequence of
any node designator in Lock List j, for any i≠j. (Interpretation:
no node named in the Lock List of one partition may be an ascendant
or descendant of any node named in the Lock List of any other partition.)

There is no necessary connection between the presence of any
node designator in any Lock List and the existence of any node in
Named System Storage or Named Partition Storage.

Initially, all Lock Lists are empty. In particular, when the
process in partition i is initiated, Lock List i is empty. A Lock
List is changed only as the direct result of one of the following
actions.
1.   Termination of the process (explicit or implicit execution
     of HALT) in partition i empties Lock List i.
2.   Execution of LOCK without arguments in partition i empties
     Lock List i.
3.   Execution of LOCK with an argument in partition i:
          a.   first, empties Lock List i;
          b.   then, sets the entire node designator list denoted
               by the argument into Lock List i, but only if
               the Descendant Exclusivity Rule will not thereby
               be violated. If the Rule would be violated,
               the Lock List remains empty.

4.3   Open List P-vector

The syntax of a "device designator" is specified by each system
implementor; however, this much can be said here.
1.   It is a data string.
2.   It uniquely designates a "device", which is a sequential
     character source and/or a sequential character sink.

Each partition of the system, numbered 1 to P, is associated
with an "Open List". The collection of these P Open Lists makes
up the "Open List P-vector" in System Storage. Each Open List may
be empty, or it may be a list of device designators separated by
the nondata separator ¢¢.

Open List 1:   [<u>device designator</u>   [¢¢ <u>device designator</u>] ...]

.
.
.

Open List P: [<u>device designator</u>   [¢¢ <u>device designator</u>] ...]

Device Exclusivity Rule
No device designator in Open List i may designate the same device
as any device designator in Open List j, for any i≠j. (Interpretation:
partitions "own" devices exclusively.)

Initially, all Open Lists are empty. In particular, when the
process in partition i is initiated, Open List i is empty. (See
the discussion of the Principal Device Convention in 5.4 for an ex-
ception.) An Open List is changed only as the direct result of one
of the following actions.
1.   Execution of OPEN in partition i adds those device designators
      to Open List i which are denoted by the argument list.
2.   Execution of CLOSE in partition i removes those device
      designators from Open List i which are denoted by the argument
      list. (See the discussion of the Principal Device Convention
      in 5.4 for consideration of a possible side effect of CLOSE.)
3.   Termination of the process (explicit or implicit execution
      of HALT) in partition i empties Open List i.

4.4   Job Number P-vector

In order for a process to execute, it must be "active". The
means by which a process becomes active is not defined in the Standard.
Once a process is active, it remains active until it executes HALT
or its equivalent (see 6.2 paragraph 2).

Each partition of the system which contains an active process
is associated with a unique, positive-integer "Job Number". Inactive
partitions have the Job Number zero. The collection of these P Job
Numbers makes up the "Job Number P-vector" in System Storage. The
Job Number of a process is assigned when the process becomes active,
and it remains fixed throughout the life of the process. Its value
is directly obtained by execution of the special variable $J as an
expression atom. At no time may two active partitions have the same
Job Number value. In some implementations the Job Number of an active
partition may simply be the partition number, but this is not necessarily
the case.

4.5   Clock

The content of this storage register is always defined and may
be directly obtained by execution of the special variable $H as an
expression atom. The clock value reflects the correct date and time
and has the form of two integers separated by a comma: D,S . D
is a day counter. S is a second counter running from 0 to 86399
for each value of D. D has a fixed value from each midnight to the
immediately succeeding midnight; at the instant of midnight D is
increased by 1 and S is set to 0. S counts seconds after midnight.
The reference clock value 0,0 is the first second of January 1, 1841.

CHAPTER 5

PARTITION STORAGE

Table of Contents

CHAPTER 5

PARTITION STORAGE

In each of the P partitions of the system Partition Storage contains the following components.
1. The Named Partition Storage data structure, commonly called "local data" or the "symbol table".
2. The Naked Indicator.
3. The Test Switch.
4. The Current Device Designator.
5. The Current Device Horizontal Cursor and Current Device Vertical Cursor.

5.1 Named Partition Storage

In each partition, Named Partition Storage is a data structure which follows the identical set of rules as Named System Storage (see 4.1), except as follows.

Rule 3.d governing contents of attribute blocks at each node is changed as follows.
The spelling of the N part of the Name attribute, instead of having the syntax $\hat{}$name , has the syntax name .

Rule 3 governing coupling between attribute block contents and tree structure reads as follows.
Each partition has its own distinct tree. Upon initiation of the process in partition i, its tree consists only of a directory node, for which D=0.

5.2 Naked Indicator

The content of the Naked Indicator is either empty ("undefined"), or it satisfies the syntax of node designator (see 4.2).

The Naked Indicator is changed only as a direct result of one of the following actions in the partition.
1. Initiation of the process causes the Naked Indicator to be undefined.
2. Execution of an unsubscripted global variable causes the Naked Indicator to be undefined.
3. Execution of a global variable or naked reference whose implied immediate ascendant either does not exist or has D=0 or 1 causes the Naked Indicator to be undefined. This will never occur, however, when the global variable or naked reference occurs in an argument of the SET command to the left of the = delimiter.
4. Except for the above, execution of a global variable or naked reference causes the content of the Name attribute of the implied immediate ascendant to be copied into the Naked Indicator. That is, the remainder after removing the last ¢ Sk-1 from the implied level k node designator is placed into the Naked Indicator.

Execution of a <u>naked reference</u> is erroneous when the Naked Indicator
is undefined. If the Naked Indicator is defined, execution of the
naked reference ^(<u>expression 1</u>,...,<u>expression i</u>), where S1 = the
value of <u>expression 1</u>, ..., Si = the value of <u>expression i</u>, implies
the node designator

> content of Naked Indicator ¢ S1 ¢ ... ¢ Si ,

after the use of which the Naked Indicator is given the value obtained
by deleting ¢ Si.

5.3 Test Switch

The content of the Test Switch register is one of three data
strings: empty ("undefined"), 0 ("false"), or 1 ("true"). When
defined, this content may be directly obtained by execution of the
special variable $T as an expression atom.

The content of the Test Switch is changed only as a direct result
of one of the following actions in the partition.
1. Execution of IF with an argument places the truth-value
   interpretation of the value of the argument into the Test
   Switch.
2. Execution of LOCK, OPEN, or READ with an argument which
   has a timeout attached to the argument implies a test of
   some condition; specifically, the ability to load the par-
   tition's Lock List without violating the Descendant Exclusivity
   Rule, the ability to load the partition's Open List without
   violating the Device Exclusivity Rule, or explicit termination
   of the input data string prior to resumption of execution
   after the timeout, respectively. If the condition is satisfied,
   the Test Switch receives the value 1; otherwise it receives
   the value 0.

In addition to the ability to obtain directly the content of
the Test Switch by execution of $T, the process may, by execution
of ELSE or IF, conditionalize execution of the remainder of the line
containing the ELSE or IF depending on the value of the Test Switch.

5.4 Current Device Designator

See the first paragraph of 4.3 for a definition of "device" and "device designator". The content of the Current Device Designator storage register is either empty ("undefined") or else it is a data string which satisfies the syntax of <u>device designator</u>. The content of the Current Device Designator may be directly obtained by executing the special variable $I as an expression atom.

The "current device" is that device which is uniquely designated within the partition for data transfer upon execution of READ or WRITE. When the designation of the current device is undefined (i.e., the content of the Current Device Designator is empty), the result of execution of READ or WRITE is undefined. Otherwise, the Current Device Designator designates the current device.

<u>Current Ownership Rule</u>
At all times the current device of partition i must also be designated in Open List i. (Interpretation: the process must own a device before it may be declared current. Also, when the process CLOSEs a device which is current, the device designator is removed from the Current Device Designator as well as from the Open List.)

The implementor has an option with respect to the initial content of the Current Device Designator and the response to CLOSE naming the current device. Following is a description of the "Principal Device Convention", which is one choice.

1. There is a device D for this process called the "Principal Device". The identity of the Principal Device is specified by the implementor and it does not change throughout the life of the process.
2. Upon initiation of the process in partition i, Open List i designates D (and D only), and partition i's Current Device Designator designates D.
3. Whenever a CLOSE is executed which would empty the Current Device Designator, a designator of D is placed into the Current Device Designator and the Open List (if it is not already there and if the operation can be performed without violating the Current Ownership Rule).

If the implementor does not follow the Principal Device Convention with respect to this process, the Current Device Designator will be empty initially and as a result of executing a CLOSE which names the current device.

Except as specified above, the content of the Current Device Designator is changed only as a direct result of execution of USE; the device designator specified by the argument of the executed USE replaces the content of the Current Device Designator, provided that the specified device designator designates a device also designated by the Open List.

5.5 Current Device Horizontal and Vertical Cursors

As part of the status information stored for each device in
System Storage are two registers per device, X and Y, called the
horizontal cursor and vertical cursor, respectively. In Partition
Storage there are two registers, CX and CY, called the Current Device
Horizontal Cursor and the Current Device Vertical Cursor, respectively,
which make available the values of X and Y for the current device.
The contents of CX and CY are directly available by execution of
the special variables $X and $Y, respectively, as expression atoms.

When the content of the Current Device Designator register changes,
the contents of CX and CY are placed into the X and Y registers of
the former current device and the X and Y registers of the new current
device are placed into the CX and CY registers. Thus, as ownership
of a device changes, the state of its cursors is not lost.

The content of each cursor is a data string which satisfies
the syntax of <u>nonnegative integer</u>. The initial value of each cursor
is not defined until the # format is executed by some process while
the device is current in that process.

Aside from the changing of CX and CY arising from changing the
current device, CX and CY are changed only by format operations and
character transfers performed during the execution of READ and WRITE.
Each format operation affects the values of CX and CY as follows.

> !   places 0 into CX and adds 1 to CY.
> #   sets CX and CY to 0.
> ?n places the value of max(n,CX) into CX.

The effects on CX and CY resulting from data transfers performed
during the execution of READ and WRITE are specified in the MUMPS
Language Standard as goals; the framers of the Standard recognized
that some implementations may exist within the constraints of operating
systems which may make absolute achievement of all the goals impractical.
The goals are stated here, and each implementor is expected to come
as close to 100% achievement as is practical.
1. The effect on CX and CY of transferring a character is
    registered when that character is transferred.
2. The effect on CX and CY of transferring a character is
    a function of each character's identity, and is independent
    of the context of the character or whether READ or WRITE
    is being executed.
3. The following effects on CX and CY are specified.

Each graphic:	Add 1 to CX.
Backspace (BS):	Set CX = max(0,CX-1).
Line Feed (LF):	Add 1 to CY.
Carriage Return (CR):	Set CX = 0.
Form Feed (FF):	Set CX = 0, CY = 0.

CHAPTER 6

THE PARTITION STACK
AND CONTROL OF EXECUTION SEQUENCE

Table of Contents

CHAPTER 6

THE PARTITION STACK
AND CONTROL OF EXECUTION SEQUENCE

The Partition Stack is a last-in, first-out data structure which
controls the sequence of execution of a MUMPS process. Each item of the
stack contains the following five components.
1. A routine body.
2. A Line Pointer which designates one line of the routine body.
It can also designate the _eor_ at the end of that routine.
3. A Line Buffer which contains a (possibly modified) copy of the
line designated by the Line Pointer.
4. A Character Pointer which designates one character of the Line
Buffer.
5. A two-valued indicator called the For Scope switch.

[Port: Routines designed for portability should not cause the Partition
Stack to be nested beyond fifteen levels.   :Port]

6.1   Normal Execution Sequence

The execution of a MUMPS process is a sequence of atomic operations,
each such atomic operation being the execution of one character.
The routine body whose characters are currently being executed is
always that at the top of the stack. Within this routine body the
sequence of characters being executed is described by rules given
in the remainder of this chapter. Normally, the following rules
apply.
1. The "normal execution sequence" prevails except when it
is altered by execution of HALT, IF, ELSE, FOR, QUIT, _eor_,
by execution of an argument of DO, GOTO, IF, or XECUTE,
or by indirection. Rules 2 and 3 describe the normal execution
sequence.
2. Execution of a given line is defined to be the sequence
of character executions which occur while a copy of that
line is in the Line Buffer. When the Line Pointer is changed
to designate a new line, the following occurs.
   a.   A copy of the new line, from the character immediately
to the right of _ls_ to and including the _eol_,
is taken from the routine body and placed into
the Line Buffer.
   b.   The Character Pointer is set to designate the
leftmost character of the Line Buffer.
3. A copy of the character designated by the Character Pointer
is obtained and executed. If the character is _eol_ and
the For Scope switch is off, its effect is to cause the
Line Pointer to advance to the next line (or to the _eor_).
If the character is not _eol_, then just prior to execution
of the character, the Character Pointer is advanced to
the next character to the right; then the character which
had been obtained is executed.

The normal execution sequence, then, is a strict left-to-right sequence of execution of characters within lines and of lines within the routine body. The identity of the first line to be executed when a new routine body is pushed onto the top of the Partition Stack is specified within the DO argument which names the routine. In the default case, i.e., when the DO argument contains no line specification or upon initiation of the process without a line specification, the first line of the routine body is implied.

In the initial state of a process just prior to execution of the first character, the Partition Stack contains one routine body, the Line Pointer and the Character Pointer designate the first character of the first executed line of the routine body, and the For Scope switch is off.

6.2   Changes of Sequence

Deviation from the normal execution sequence can occur in the following ways.

1. Execution of HALT causes the process to terminate. Termination of a process implies the following operations in System Storage associated with the partition k in which the process terminates.
   - a. The kth element of the Job Number P-vector is set to zero.
   - b. The kth element of the Lock List P-vector is made empty.
   - c. The kth element of the Open List P-vector is made empty.

2. Execution of QUIT with the For Scope switch off, or execution of eor, causes the top element of the Partition Stack to be popped. If the Partition Stack thereby becomes empty, the process terminates as if HALT had been executed. If there is still a routine body on the Partition Stack, the next character executed is the character designated by the Character Pointer in the new stack top.

3. Execution of any of the following subject to the condition stated causes the Character Pointer to scan to the right fetching but not executing each character until it points to the eol of the Line Buffer; that is, the effect of executing any of the following is to suppress execution of the remainder of the line to its right.
   - a. Execution of argumentless IF such that the content of the Test Switch is 0.
   - b. Execution of ELSE such that the content of the Test Switch is 1.
   - c. Execution of an IF argument whose value (after taking the truth-value interpretation) is zero.

4. Execution of a DO argument causes a new element to be pushed onto the Partition Stack. The procedure is described as follows.

    a. If the DO argument contains a routine name.

      (1) The routine body is that designated by the routine name.

      (2) The Line Pointer is set to point to the line designated by the DO argument; if the DO argument contains no line reference, the Line Pointer is set to point to the first line of the routine body.

      (3) The Line Buffer is loaded and the Character Pointer is set to point to its leftmost character.

      (4) The For Scope switch is set off.

    b. If the DO argument contains no routine name.

      (1) The routine body is a copy of the one at the top of the stack.

      (2) The Line Pointer is set to point to the line designated by the DO argument. Then (3) and (4) above are executed.

5. Execution of a GOTO argument causes the following two operations to occur in sequence.

    a. The top element of the stack is popped as if QUIT with the For Scope switch off had been executed.

    b. An element is pushed onto the stack as if a DO argument with the same spelling as the GOTO argument had been executed.

6. Execution of an XECUTE argument is similar to execution of a DO argument with the following amendments.

    a. The routine body is a copy of the one at the top of the stack.

    b. The Line Pointer is set to point to the last line of this routine body, just preceding the <u>eor</u>.

    c. The Line Buffer is loaded with the following string

        value of XECUTE argument   <u>eol</u>

    and the Character Pointer is set to its leftmost character.

7. Execution of the command word FOR causes the For Scope
switch to be turned on, regardless of its prior state.
After evaluation of all expressions in the first for parameter,
a unique marker character, called a Parameter Marker,
is inserted into the Line Buffer between the for parameter
just evaluated and the separator immediately to its right.
The Character Pointer is then moved to the right to the
first character to the right of the space following the
for parameter list. (If the for parameter list is immediately
followed by eol, the Character Pointer points to this eol.)
Another unique character, called a Scope Marker, is inserted
immediately to the left of this character, and normal execution
resumes. The following subsequent breaks in the normal
sequence can occur.

    a. If eol is executed with the For Scope switch
on and the scope must be executed again under
the current for parameter, the Character Pointer
repositions at the character to the right of
the rightmost Scope Marker, and the normal execution
sequence resumes.

    b. If eol is executed with the For Scope switch
on and the scope does not have to be executed
again under the current for parameter, the Character
Pointer scans to the left looking for a Parameter
Marker.

        (1) If a Parameter Marker is found and the character
immediately to its right is a space or eol
(that is, the scope of the FOR under control
of the last for parameter has just been
executed), then the Parameter Marker and
the Scope Marker to its right are deleted
and the leftward scan described in b. above
is resumed.

        (2) If no Parameter Marker is found, the For
Scope switch is turned off and the normal
eol execution described in 6.1 paragraph
3. above occurs.

        (3) If a Parameter Marker is found and the character
immediately to its right is a comma, the
Parameter Marker is deleted, the for parameter
to its right is evaluated, the Parameter
Marker is inserted immediately to the right
of this for parameter, the Character Pointer
is moved to point to the character immediately
to the right of the Scope Marker, and normal
execution resumes.

    c. Execution of QUIT with the For Scope switch on
causes the Character Pointer to scan to the left
until it encounters a Parameter Marker. This
Parameter Marker and the Scope Marker to its
right are deleted. Then the eol procedure beginning
at 7a. above is executed.

    d. Execution of a GOTO argument with the For Scope
switch on is unchanged from that described in
5.

8. Execution of @ (signaling indirection) causes the following steps to occur.

    a.   The Character Pointer continues to the right as the expression atom to the right of the @ is evaluated. Note that this evaluation can encounter a @, so the whole of 8. is recursive.

    b.   After the expression atom is evaluated, the value of the expression atom is inserted into the Line Buffer immediately to the right of the expression atom, and the Character Pointer is set to point to the leftmost character of this value. (If the value is empty, the Character Pointer points to the first character to the right of the expression atom, and the effect of the following step is null.)

    c.   As each character in the Line Buffer which arose from the evaluation of the expression atom is designated by the Character Pointer and a copy of it is fetched, that character is deleted from the Line Buffer. The effect is to leave no trace that indirection has occurred so that a single instance of indirection may be called repeatedly in the scope of a FOR.

CHAPTER 7

ROUTINES AND LINES

The routine is the unit of algorithm interchange. Programs and packages
are unordered collections of routines. When called for execution, a routine
is denoted by its routine name. The syntax observed by routine names
is the same as name.

$$\text{routine name} ::= \left|\begin{array}{c} \% \\ \\ \text{alpha} \end{array}\right| \left[\begin{array}{c} \text{digit} \\ \\ \text{alpha} \end{array}\right] \quad \ldots$$

[Port: Routine names containing 9 or more characters are identified and
distinguished only on the basis of their left 8 characters. No lower-case
alphabetic characters may occur in a routine name. :Port]

When one refers to the syntax of a routine, one is referring to the
routine as stored on a linear external medium as in algorithm interchange.
In this context, a routine consists of two parts: the head, which contains
the routine name for identification in interchange, and the body, which
contains the executed code, consisting of a sequence of lines.

routine   ::=   routine head   routine body

routine head   ::=   routine name   eol

routine body   ::=   line [line] ... eor

In interchange, the three separators ls, eol, and eor denote the
following ASCII character or control character pairs.

ls   ::=   SP

eol   ::=   CR LF

eor   ::=   CR FF

Thus, if the routine is printed on a device which observes ASCII
control conventions, the routine name would appear as the first printed
line and the executed code would appear on the remaining printed lines.

Analogously to the structure of the routine, each line consists of
two parts. The line head may contain an optional label for identifying
the line, and the line body contains the executable commands, if any,
of the line.

line ::= line head line body

line head ::= [label] ls

line body ::= [command [_ command] ... [_ comment]] eol
                                comment

label ::= | name |
          | integer literal |

comment ::= ; [graphic] ...

[Port: The total number of characters in a line, excluding the eol,
but including the remainder of the line body and the whole of the line
head, may not exceed 255. :Port]

Labels are used to identify lines. Line references occur in DO and
GOTO commands and in the $TEXT function. DO and GOTO may refer to a line
in any routine, whereas $TFXT refers only to the routine body at the top
of the Partition Stack.

Within a given routine, a line whose line head contains a label may
be designated by naming the label. Thus the simplest form of a line reference
is an occurrence of label. More generally, any line in a given routine
may be designated by referring to a prior line containing a label in its
line head. The form label + integer expression (where i is the integer
interpretation of the value of the expression) means "the ith line after
the line containing label in its line head". The reference label + 0
is equivalent to label. Negative values of integer expression are not
allowed.

Within any given routine, no spelling of any label may occur more
than once in any line head. When the label consists entirely of numeric
digits, leading zeros are significant to its spelling; that is, the label
01 is different from the label 1. [Port: Labels containing 9 or more
characters are identified and distinguished only on the basis of their
left 8 characters. :Port]

In the general form of a line reference, indirection is permitted on the label portion but not on the line reference as a whole.

line reference   ::=   dlabel [+ integer expression]

dlabel   ::=
| label |
| @ expression atom V dlabel |

The $TEXT function has a default definition which handles the case of a line reference either reaching beyond the end of a routine or referring to a label not appearing in a line head in the routine. DO and GOTO do not have such default definitions; therefore the following conditions are not allowed in the context of DO or GOTO.

1.   A spelling of label which does not occur in a line head of the routine denoted.
2.   A value of integer expression so large as not to denote a line within the routine body.

DO and GOTO can specify a routine as well as a line, and their arguments therefore contain a more general form of line reference called an entry reference. Within an entry reference, three syntactic cases may be distinguished.

1.   The entry reference has the form of line reference , in which case the routine body implied is that at the top of the routine stack.
2.   The entry reference has the form of ^routine reference , in which case the first line of the routine whose name is denoted by the routine reference is implied.
3.   The entry reference has the form of line reference^routine reference , in which case the designated line of the designated routine is implied.

entry reference   ::=
| line reference [^ routine reference] |
| ^ routine reference |

routine reference   ::=
| routine name |
| @ expression atom V routine reference |

CHAPTER 8

COMMANDS

Table of Contents

CHAPTER 8

COMMANDS

8.1 Introductory Definitions

8.1.1 Command Syntax

In Chapter 7 it is shown how the syntactic object command occurs within a line body. In this chapter the syntax and semantics of command are made explicit.

In place of any occurrence of command in the definition of line body can stand any of the command definitions of 8.2. Every command begins with a mnemonic command word spelled entirely with capital letters, such as SET.

Commands fall into two basic categories. Some command words have variations which fall into both categories.

1. Argumentless commands. The syntax of an argumentless command consists of the command word, possibly followed by a space.
   a. If the occurrence of an argumentless command immediately precedes the eol, there is no space following the command word.

   QUIT eol

   b. If the occurrence of an argumentless command does not immediately precede the eol, the command word is followed by exactly two spaces, the first considered to be part of the command, and the second considered to be between this command and the next command or comment. (Refer to the syntax of line body in Chapter 7.)

   ELSE ⊔ ⊔ GOTO ⊔ X eol

2. Commands with arguments. The syntax of a command which takes an argument consists of the command word followed by a single space followed by an argument or argument list.

   IF ⊔ X=Y ⊔ DO ⊔ A ⊔ QUIT:X=1 ⊔ ⊔ DO ⊔ B eol

8.1.2 Format of Command Definitions

Each command definition of 8.2 is organized into the following parts.

1. <u>Syntax</u>. The syntax of the most suggestive, and usually the simplest, form of the command is given here.
2. <u>Elaboration</u>. Variations on the syntax given in 1. are discussed. These can fall into the following five categories discussed in detail in 8.1.3.
   - a. Abbreviation of the command word
   - b. Post-conditionalization of the command word
   - c. Listing of multiple arguments
   - d. Post-conditionalization of any argument
   - e. Argument indirection
3. <u>Execution</u>. The effect of executing the command is described.
4. <u>Cross-reference</u>. Reference is made to other portions of this reference manual where a related discussion may shed light on the definition of this command.
5. <u>Examples</u>. Examples of use of the command are given, usually with explanation and in a sequence which permits the reader to make useful inferences about the command.

8.1.3 Elaborations

8.1.3.1 Abbreviation of the Command Word

Wherever a command word may properly occur, it may occur in either of two forms: as fully spelled out in the syntax portion of the command definition, or in an initial-letter abbreviation. Other forms are not allowed. For example,

$$GOTO \sqcup A$$

may be written

$$GOTO \sqcup A$$

or

$$G \sqcup A$$

but neither

$$GO \sqcup A$$

nor

$$GOT \sqcup A$$

nor

$$GOING \sqcup A$$

nor

$$GOTOHERE \sqcup A \qquad .$$

8.1.3.2   Post-conditionalization of the Command Word

A post-conditional is a colon immediately followed by a truth-valued expression, the pair of which can appear as an optional suffix to certain command words or command arguments.

post-conditional   ::=   : truth-value expression

All command words defined by the MUMPS Standard except ELSE, FOR, and IF may be post-conditionalized, either in their spelled-out or abbreviated forms.   For example, a post-conditionalized form of

SET ⎵ X=1

is

SET:Y=2 ⎵ X=1

or

S:Y=2 ⎵ X=1   .

(Note the two different uses of the "=".)

The interpretation of the post-conditional attached to a command word is as follows.
1.   If the post-conditional is absent, the command as a whole is executed unconditionally, subject to external factors which might inhibit its execution, such as a preceding ELSE or IF.
2.   If the post-conditional is present, it is evaluated before anything to its right is executed.   If the truth-value interpretation of its value is 1 (true), then the command as a whole is executed. If the truth-value interpretation of the value of the post-conditional is 0 (false), then no part of the command is executed and no argument is evaluated.

The evaluation of a post-conditional can have side effects even if it inhibits the execution of its host command.   For example,

SET:$D(^(1,2))=3 ⎵ X=X+1

will never change X but will always change the Naked Indicator.

Since the post-conditional does not affect the Test Switch, its use is not equivalent to the use of IF.

8.1.3.3   Listing of Multiple Arguments

The syntax presented in 8.2 for each command which accepts an argument has the following structure.

command word   a single space   a single argument

If the elaboration does not state explicitly that arguments may be listed, then the argument form shown describes all permissible argument structures for this command. If the elaboration states that arguments may be listed, then the following convention applies.

Let Y denote the command word, and let A1, A2, ..., An each denote a single argument, each with syntax as specified in the syntax description. If the syntax description is shown as

Y ⌴ A1

then the following forms of the command syntax are also permitted.

Y ⌴ A1,A2

Y ⌴ A1,A2, ... ,An   .

(The three dots are not to be taken literally but suggest indefinite but finite repetition, with a comma and no space between each pair of arguments.)

The interpretation of

Y ⌴ A1,A2, ... ,An

is precisely the same as

Y ⌴ A1 ⌴ Y ⌴ A2 ⌴ ... ⌴ Y ⌴ An   .

That is, each comma separating command arguments preceded by the command word Y is literally to be interpreted as

⌴ Y ⌴   .

[Port: The restriction on the number of characters in a line applies as the line actually appears in routine interchange, and not as the line might appear after making each substitution of ⌴ Y ⌴ for , :Port]

8.1.3.4   Post-conditionalization of Any Argument

Only the commands DO, GOTO, and XECUTE permit post-conditionalization of arguments; therefore the following discussion applies only to these three commands.

Any argument may be post-conditionalized, independently of the post-conditionalization of the command word or of any other argument in the argument list.   The interpretation of a post-conditionalized argument is as follows.

1.   In what follows, the command word may be assumed to be not post-conditionalized, for this reason.   If a post-conditional on the command word is present and (its value is) false, then no argument interpretation is necessary, since no argument of the command is interpreted or executed.   If a post-conditional on the command word is present and true, then each argument is interpreted in turn as if the command word had not been post-conditionalized.

2.   Let Pi denote a post-conditional appearing on argument Ai. With respect to the question of whether or not the explicit execution of a particular argument occurs,

        Y ␣ A1:P1

is interpreted as

        Y:P1 ␣ A1   .

In the interpretation of the expansion of

        Y ␣ A1,A2, ... ,An

if any argument is of the form

        Ai:Pi   ,

then in the expansion in which the commas are removed it appears as

        Y:Pi ␣ Ai   .

3. With respect to side effects, the matter is complicated by the
    fact that a post-conditional has no effect on the normal execution
    sequence (see Chapter 6) until after it, and the argument which
    it qualifies, have already been evaluated. Therefore, in the
    execution of

    DO ␣ X:A

    the entry reference X and the expression A are always fully
    evaluated (unless the DO is falsely post-conditionalized) even
    if A is false and therefore the subroutine call is not performed.
    Therefore,

    XECUTE ␣ ^(3,4):0

    will have the effect of attempting to change the Naked Indicator
    and of attempting to fetch the designated value (each of which
    attempts can lead to an error if the respective arguments are
    not defined), but not of executing the evaluated expression.

8.1.3.5   Argument Indirection

The following discussion applies to those commands for which argument
indirection is permitted. Any command admitting argument indirection
also admits argument listing.

The indirection call

@ _expression atom_

may occur in place of any argument in an argument list, or in place of
any isolated argument. Thus, where the syntax permits

Y ␣ A1

then

Y ␣ @ _expression atom 1_

is permitted, and where the syntax permits

Y ␣ A1,A2, ... , An

then

Y ␣ A1,A2, ... ,@ _expression atom i_, ... ,An

is permitted.

The value of <u>expression atom i</u> must be an argument or argument list satisfying the syntax specification of Y. Thus, the value of <u>expression atom 1</u> in Y _ @ <u>expression atom 1</u> may be a single argument

        A1

or an argument list

        A1,A2, ... , An

or an argument list containing one or more indirection calls

        A1,A2, ... ,@ <u>expression atom i</u>, ... ,An

requiring subsequent indirection evaluation. These three possibilities are equally valid for an @ <u>expression atom i</u> occurring anywhere in an argument list, since the expansion of

        Y ⎵ A1,A2, ... ,@ <u>expression atom i</u>, ... ,An

is

      Y ⎵ A1 ⎵ Y ⎵ A2 ⎵ ... ⎵ Y ⎵ @ <u>expression atom i</u> ⎵ ... ⎵ Y ⎵ An   .

8.1.4   Some Common Elements Among Command Definitions

8.1.4.1   Formats in READ and WRITE

The format, when executed as part of a READ or WRITE command, generates format-controlling output operations. The parameters of a format are executed one at a time, in left-to-right order.

$$\text{format} \quad ::= \quad \left|\; \left|\begin{matrix} ! \\ \# \end{matrix}\right| \; \left[\begin{matrix} ! \\ \# \end{matrix}\right] \quad \cdots \quad [\; ? \; \underline{\text{integer expression}} \;] \atop ? \; \underline{\text{integer expression}} \;\right|$$

The format-control parameters may take the following forms.

! causes a "new line" operation to be executed at the current device. (See 5.4 for a discussion of the current device.) The effect of ! is equivalent to writing CR LF on a pure ASCII device. In addition CX is set to 0 and 1 is added to CY (see 5.5).

\# causes a "top of form" operation to be executed at the current device. The effect of \# is equivalent to writing CR FF on a pure ASCII device. In addition, CX and CY are set to 0. When the current device is a display, the screen is blanked and the cursor is positioned at the upper-left corner.

? integer expression produces an effect similar to "tab to column integer expression". More precisely, if CX is greater than or equal to the value of integer expression, there is no effect. Otherwise, the effect is the same as writing (integer expression - $X) spaces. ($X is the value of CX.) Note that the leftmost column of a line is given the number 0.

If the current device does not accept output, execution of a format has no effect, except as described on CX and CY.

8.1.4.2 Timeouts in LOCK, OPEN, and READ

The LOCK, OPEN, and READ commands can make use of an optional timeout specification, invoked by the presence of the timeout in the executed command.

timeout ::= : numeric expression

Each of these three commands has associated with its definition a specific external condition to which the execution of the command is sensitive.

LOCK   Execution of LOCK may not proceed in violation of the Descendant
            Exclusivity Rule (see 4.2). The condition tested is the absence
            of any descendancy relation between each node designator implied
            by the argument of the LOCK command and the union of Lock Lists
            over all other partitions of the system.

OPEN   Execution of OPEN may not proceed in violation of the Device
            Exclusivity Rule (see 4.3). The condition tested is the absence
            of any common device between any device designator implied by
            the argument of the OPEN command and the union of Open Lists
            over all other partitions of the system.

READ   By definition, execution of READ is complete when the input
            message has been terminated. The condition tested is termination
            of the input message.

If the optional timeout is absent from the command argument, execution of the command will proceed if and only if the condition associated with the command is satisfied. If the condition is not satisfied, execution will be suspended until the condition is satisfied; then execution will proceed.

If the optional timeout is present in the command argument, a non-negative value of numeric expression is required. If the value of numeric expression is negative, zero is used. Let t be this nonnegative value; the timeout specifies a suspension of execution of no more than t seconds, as described in the following.

If $t = 0$, the condition is tested. If it is true, the Test Switch is set to 1; otherwise, the Test Switch 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, the Test Switch is set to 1; otherwise, the Test Switch is set to 0.

If the optional timeout is present and the rules stated above require that execution must proceed without the condition having been satisfied, resolution is obtained in the following ways, depending on the command involved.

LOCK   The function specified by the argument in question is only partly executed (that is, the partition's Lock List is left empty) and control proceeds to the next command or argument.

OPEN   The function specified by the argument in question is not executed and control proceeds to the next command or argument.

READ   Whatever input characters have arrived prior to resumption of execution are taken as the whole input message, even if that message is empty. The disposition of those input characters which arrive after resumption of execution but prior to execution of the next READ, if any, is specified by the implementor.

Syntax

```
          |[ ⌴ ]                                           |
   BREAK  |                                                |
          | ⌴ argument syntax specified by implementor |
```

Elaboration

1. The command word may be abbreviated to the single letter B.
2. The command word may be post-conditionalized.
3. The implementor specifies whether there is a form with arguments and, if so, whether arguments may be listed.
4. The implementor specifies whether arguments may be post-conditionalized.
5. The implementor specifies whether argument indirection is permitted.

Execution

1. _Argumentless form._ No change is made to partition storage or system storage as a direct result of execution of BREAK. Execution is suspended until receipt of a signal, whose nature is specified by the implementor, from a device, whose identity is specified by the implementor. (It is possible that storage values may be changed as a result of external intervention while execution is suspended.)

2. _Form with argument._ Execution is specified by the implementor.

Examples

1. B   Unconditional BREAK

2. B:X=3   BREAK if X=3

3. SET X=^A(3) IF X=0 BREAK   KILL ^A

                                 A BREAK is inserted before a global
                                 is deleted so that the programmer
                                 may examine the job's status.

CLOSE

Syntax

> CLOSE ⌴ device specifier [: device parameters]
>
> device specifier   ::=   expression
>
> device parameters   ::= | expression |
> | (expression [: [expression]] ...) |

Elaboration

1. The command word may be abbreviated to the single letter C.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The value of the Open List corresponding to this partition is subject to change. Depending on whether the value of the device specifier designates the current device, the content of the Current Device Designator is subject to change.

The value of the expression in the device specifier is a device designator which denotes a device, or it may not, depending on a specification provided by the implementor. If the device designator implied by the argument designates the same device as any device designator in the Open List, then each of the latter is removed from the Open List. See the discussion of device parameters under OPEN (8.2.12).

Cross-reference

4.3 Open List and device designator
5.4 Current Device Designator and Principal Device Convention

Examples

1. C 3   Device 3 is CLOSEd.

2. C:X=1 X   Device X is CLOSEd only if X=1.

Syntax

DO ␣ <u>entry reference</u>

Elaboration

1. The command word may be abbreviated to the single letter D.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may be post-conditionalized.
5. Argument indirection is permitted.

Execution

Execution of an argument of DO is described in full in terms of the system model in 6.2 paragraph 4. Execution is equivalent to a subroutine call. The entry point of the subroutine is defined by the argument. The exit point occurs upon execution of QUIT not in the scope of FOR, or of <u>eor</u>, neither of which is serving as an exit point for a subsequently executed DO or XECUTF.

Cross-reference

6.2 paragraph 4   Fxecution description
6.2 paragraph 2   Fxecution of QUIT not in the scope of FOR, and of <u>eor</u>
Chapter 7   Syntax and interpretation of <u>entry reference</u>

Examples

1. DO X WRITE Y   The subroutine beginning at the
                                     line labeled X is executed; then
                                     the variable Y is written.

2. DO X,Y   This is equivalent to DO X DO Y

3. DO ^PGM WRITE Y   Routine PGM is invoked; then Y is
                                     written.

4. DO A,^PGM,^ROU   Multiple routine names and labels
                                     may be listed as arguments.

5. DO INT^JTO   Routine JTO is invoked beginning
                                     at label INT.

6. D:X=1 A^PGM,B,^TEST   The list of subroutines is invoked
                                     only if X=1.

7. DO ^CEN:'A,B   Routine CEN is invoked if A=0;
                                     subroutine B is invoked independently
                                     of the value of A

8. D @X   Indirection may be used in the
    D @X^PGM   DO command
    D ^@XYZ
    D @A^@B
    D:@A=B @C^@D:@E=@F

Syntax

        ELSE [_]

Elaboration

    1.   The command word may be abbreviated to the single letter E.
    2.   The command word may not be post-conditionalized.

Execution

    Execution of ELSE is described in terms of the system model in 6.2
    paragraph 3.   When ELSE is executed and the Test Switch ($T) has
    the value 1, the execution of the remainder of the line up to, but
    not including, the eol is inhibited.   In the scope of FOR the effect
    of each execution of a given ELSE is considered separately at the
    time it is executed.

Cross-reference

    6.2 paragraph 3   Execution description
    5.3   Test Switch

Examples

    The following two examples produce the same effect.

    IF X=1 WRITE "YES"
    ELSE   WRITE "NO"

    WRITE:X=1 "YES" WRITE:'(X=1) "NO"

FOR

<u>Syntax</u>

FOR ⌴ <u>local variable</u> = <u>for parameter</u> [, <u>for parameter</u>] ...

<u>for parameter</u> ::= <u>expression</u>

<u>start-step parameter</u> : <u>numeric expression 3</u>

<u>start-step parameter</u>

<u>start-step parameter</u> ::= <u>numeric expression 1</u> : <u>numeric expression 2</u>

<u>Elaboration</u>

1. The command word may be abbreviated to the single letter F.
2. The command word may not be post-conditionalized.
3. Arguments may not be listed; however, <u>for parameters</u> may be listed in any order within the single argument.
4. Arguments may not be post-conditionalized.
5. Argument indirection is not permitted, nor is any indirection permitted at the level of the <u>for parameter</u>.

<u>Execution</u>

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

FOR specifies and controls repeated execution of its scope for successive values of the named local variable. Each for parameter causes the local variable to be given a specified sequence of values, and the scope to be executed once for each of these values; successive for parameters control this process in turn, in left-to-right order. Any expression occurring in the <u>local variable</u>, for example in a subscript or in indirection, is evaluated once per execution of the complete FOR, prior to the first execution of any for parameter.

The following descriptions specify how each of the <u>for parameter</u> forms specifies and controls the values of the local variable and the sequence of executions of the scope. The variable names A, B, and C are to be considered hidden storage registers used solely for control of each FOR command.

1. If the for parameter is of the form **expression.**

    a. Set <u>local variable</u> = <u>expression.</u>
    b. Execute the scope once.
    c. Execution of this for parameter is complete.

2. If the for parameter is of the form <u>start-step parameter.</u>

    a. Set A = <u>numeric expression 1.</u>
    b. Set B = <u>numeric expression 2.</u>
    c. Set <u>local variable</u> = A.
    d. Execute the scope once.
    e. Set <u>local variable</u> = <u>local variable</u> + B.
    f. Go to d.

    Note that this procedure specifies an endless loop. Termination
    of this loop must occur by execution of a QUIT or GOTO within
    the scope. These two termination methods are available within
    the scope of any FOR, and they will be described in general
    below. Note also that no for parameter to the right of one
    of this form can ever achieve control.

3. If the for parameter is of the form
    <u>start-step parameter</u> : <u>numeric expression 3</u>
    and the step parameter (<u>numeric expression 2</u>)
    has a nonnegative value.

    a. Set A = <u>numeric expression 1.</u>
    b. Set B = <u>numeric expression 2.</u>
    c. Set C = <u>numeric expression 3.</u>
    d. Set <u>local variable</u> = A.
    e. If <u>local variable</u> > C, execution of this for parameter
        is complete.
    f. Execute the scope once.
    g. If <u>local variable</u> > C-B, execution of this for parameter
        is complete.
    h. Set <u>local variable</u> = <u>local variable</u> + B.
    i. Go to f.

4. If the for parameter is of the form
   start-step parameter : numeric expression 3
   and the step parameter (numeric expression 2)
   has a negative value.

   a. Set A = numeric expression 1.
   b. Set B = numeric expression 2.
   c. Set C = numeric expression 3.
   d. Set local variable = A.
   e. If local variable < C, execution of this for parameter
      is complete.
   f. Execute the scope once.
   g. If local variable < C-B, execution of this for parameter
      is complete.
   h. Set local variable = local variable + B.
   i. Go to f.

If there is more than one FOR in a line, their executions may be
considered to be nested, and a FOR to the right is said to be "within"
or "inside" a FOR to the left. When two FORs are so nested, one
execution of the scope of the outer FOR encompasses one execution
of the entire inner FOR command, corresponding to a complete pass
through the inner FOR's for parameter list (subject to early termination
by a GOTO or QUIT).

Note that any particular execution of any command in the scope of
a FOR is under control of exactly one for parameter of that FOR at
the time of the command's execution. Two commands, GOTO and QUIT,
have special effects when they are executed in the scope of a FOR.
These effects are considered below.

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

a. It terminates this particular execution of the scope at the
   QUIT; commands to the right of the QUIT are not executed during
   this execution of the scope.
b. With respect to the innermost FOR in whose scope the QUIT occurs,
   it causes the remaining values specified by the controlling
   for parameter, and all values specified by any remaining for
   parameters in the same for parameter list, not to be calculated,
   and the scope not to be executed under their control.

In other words, execution of QUIT in the scope of a FOR effects the
immediate termination of execution of the innermost such FOR. If
there is a next outer FOR, execution of this QUIT effects the immediate
termination of one execution of the scope of this FOR.

Execution of a GOTO within the scope of a FOR effects the immediate
termination of all FORs to the left of the GOTO in the same line,
and it transfers control to the point specified.

Cross-reference

6.2 paragraph 7   Execution of FOR
6.2 paragraph 2   Execution of QUIT in the scope of FOR
6.2 paragraph 5   Execution of GOTO in the scope of FOR
8.2.6   Definition of GOTO
8.2.13   Definition of QUIT

Examples

1.   F I=1:1 WRITE I Q:I>2   W "*"   The output will be 1*2*3

2.   F I=1:1:3 W I   The output will be 123

3.   F I=3:-2:-2 W I   The output will be 31-1

4.   F I=5,3.4,7,9 W I   The output will be 53.479

5.   F I=.4,1:2:5,9,10:1 DO A IF I>15 QUIT
                                  A will be executed 12 times

6.   F I=1:1:2 F J=2:2:6 W I,"@",J,"*"
                                  The output will be
                                  1@2*1@4*1@6*2@2*2@4*2@6*

7.   F I=1:1:3 F J=10:10:30 Q:I*J>30   W I,"@",J,"*"
                                  The output will be
                                  1@10*1@20*1@30*2@10*3@10*

8.   F I=3:5:0 WRITE I   Nothing will be written. The final
                                  value of I will be 3.

9.   FOR I=.01:.0001:.02 D A   This will execute A 101 times.

10.   F I=X:Y:Z D A

11.   F I=1:2:10 SET I=I-1 W I   Changing the value of the local
                                  variable which is used as an index
                                  is permitted. The output will be
                                  0123456789

```
12.  SET Z=10 F I=2:2:Z SET Z=Z-1 DO A
                                    Because the ending value of the
                                    loop is computed before the scope
                                    is executed, A will be executed
                                    5 times

13.  F I="TEST",X,3:4:5 ...          When the simple form of the for
                                    parameter is used, the index may
                                    be given nonnumeric values.

14.  F I=X:Y:Z F J=A:B:C F K=1:1:3 ... G LCS
                                    The GOTO exits from all nested FORs.

15.  F @A=1:1:3 DO B                 Although indirection is not permitted
                                    on the for parameter as a whole,
                                    it may be used as usual in place
                                    of variable names.

16.  F I=@X:1:3 DO @B

17.  F @I=@X:@Y:@Z DO @B
```

GOTO

Syntax

GOTO ⌴ <u>entry reference</u>

Elaboration

1. The command word may be abbreviated to the single letter G.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may be post-conditionalized.
5. Argument indirection is permitted.

Execution

Execution of an argument of GOTO is described in full in terms of
the system model in 6.2 paragraph 5. Execution is equivalent to
a simple transfer of control. If the executed argument names another
routine, the routine body at the top of the partition stack is replaced
by the named routine body. Note that, in contrast to DO, when arguments
are listed, at most one argument actually transfers control. When
GOTO is executed in the scope of FOR, all FORs to the left of the
GOTO in the same line are terminated with the transfer of control.

Cross-reference

6.2 paragraph 5   Execution description
Chapter 7   Syntax and interpretation of <u>entry reference</u>
8.2.5   Execution of GOTO in the scope of FOR

Examples

1.   GOTO XYZ

    Execution continues at the start of the line labeled XYZ in this routine.

2.   GOTO ^PGM

    Context switches from the routine containing the GOTO to the first line of the routine PGM.

3.   G CC+2^N1H

    Context switches to routine N1H and execution begins at the second line after the line labeled CC.

4.   G 13^GRAY:A=1

    If and only if A=1, context switches to routine GRAY and execution begins at the line labeled 13.

5.   G:A=1 13^GRAY

    Equivalent to Example 4.

6.   G ^ADM:A=0,X^CEN

    If A=0, context switches to routine ADM and execution begins at its first line; otherwise, context switches to routine CEN and execution begins at the line labeled X.

7.   G:A=1 AAA:B=2,13:D=4

    This line is equivalent to
G AAA:A=1&(B=2) G 13:D=4&(A=1) .

8.   G @X
    G ^@B
    G A^@E
    G @A^B
    G @A^@B

    There are many opportunities to use indirection within the GOTO command

9.   G:X=@Y @A+@C+D^@PGM:@E=@F

    If control transfers, the line at which execution continues is the @C+Dth line after label @A .

Syntax

HALT [_]

Elaboration

1. The command word may be abbreviated to the single letter H. Note that when HALT is abbreviated, the absence of an argument distinguishes it from HANG.
2. The command word may be post-conditionalized.

Execution

Execution of the process in this partition terminates. System storage associated with this partition is left in the following state.

1. The element of the Job Number P-vector corresponding to this partition is zero.
2. The element of the Lock List P-vector corresponding to this partition is empty.
3. The element of the Open List P-vector corresponding to this partition is empty.

Cross-reference

6.2 paragraph 1   Execution of HALT

Examples

1.	H	Unconditional HALT
2.	F:A=1 WRITF "TEST"	If A=1, the process HALTs. Otherwise, execution continues and TEST is written.
3.	H:A=@B	Indirection may be used in the post-conditional.

HANG

Syntax

>   HANG ⌴ integer expression

Elaboration

1.   The command word may be abbreviated to the single letter H.
2.   The command word may be post-conditionalized.
3.   Arguments may be listed.
4.   Arguments may not be post-conditionalized.
5.   Argument indirection is permitted.

Execution

Let t be the integer interpretation of the value of the argument. If t is zero or negative, HANG is of no effect. If t is positive, execution is suspended for t seconds. No storage contents are changed as a direct result of execution of HANG.

Cross-reference

Examples

1.   A IF $P($H,",",2)<X HANG 3 G A

If the number of seconds after midnight is less than X, execution hangs three seconds and the line repeats.

2.   H:I=3 10 WRITE !

If I=3, a ten-second HANG is executed before the WRITE command is executed.

3.   H @X
     H:$E(@A,B,@C) X+@Y+Z

Indirection may be used in the HANG command.

Syntax

Argumentless form: IF [⎵]

Form with argument: IF ⎵ <u>truth-value expression</u>

Elaboration

1.   The command word may be abbreviated to the single letter I.
2.   The command word may not be post-conditionalized.
3.   Arguments may be listed.
4.   Arguments may not be post-conditionalized.
5.   Argument indirection is permitted.

Execution

Execution of both forms of IF may be viewed in a unified way as follows.
Execution consists of two steps. The first step is not executed
by the argumentless form. The second step is common to both forms.

Step 1.   The truth-value interpretation of the value of the argument
          is placed into the Test Switch.
Step 2.   If the content of the Test Switch is 1, there is no action;
          that is, execution continues at the next command or argument.
          If the content of the Test Switch is 0, execution of every
          character in this line to the right of this argument, up
          to but not including the <u>eol</u>, is inhibited. There can
          be no direct or side effect arising from any such inhibited
          execution.

Cross-reference

Examples

1.	I X=1 WRITE X	The code to the right of the IF command is executed only if X=1.
2.	I Y S Z=3	The code to the right of the IF command is executed only if the numeric interpretation of the value of Y is nonzero.
3.	I X=1,Y=2 DO 3	Either of the two following forms will or will not DO 3 under the same conditions. IF X=1 IF Y=2 DO 3 IF X=1&(Y=2) DO 3 That is, the effect on the sequence of execution of listing arguments is similar to an implied _and_.
4.	I X,^A(Y) G 3	The side effects, however, may be different. This code is not equivalent to I X&(^A(Y)) G 3 since the latter will always evaluate ^A(Y) while the former might not.
5.	I X?1N2A!(Y=2)	Complex expressions occur frequently in IF commands
6.	IF X=1 W !,"TEST" ELSE W !,"EXAM" IF W " ARGUMENTLESS"	If X=1, this code will write TEST ARGUMENTLESS otherwise, it will write EXAM.
7.	IF @X IF A,@B,C IF A=@B,@C=D,E=F+@G+H	Indirection may be used in the IF command

KILL

Syntax

    1. "Kill All" argumentless form: KILL [⌴]

    2.   "Selective Kill" form: KILL ⌴ <u>storage reference</u>

    3.   "Exclusive Kill" form: KILL ⌴ (<u>name</u> [, <u>name</u>] ...)

Elaboration

    1. The command word may be abbreviated to the single letter K.
    2. The command word may be post-conditionalized.
    3. Arguments may be listed. In an argument list the two argument forms may be mixed. (Note that the Exclusive Kill syntax form contains only one argument.)
    4. Arguments may not be post-conditionalized.
    5. Argument indirection is permitted.

Execution

The execution of all three forms may be defined in terms of a single atomic operation: the killing of a single variable. Let V be a storage reference. Let N be the corresponding node, if any, in Named Partition Storage or Named System Storage, respectively, for which the Name attribute corresponds to V. (See 12.4 for a discussion of this correspondence.)

Killing V has the following effect. If N does not exist, there is no effect. If N exists, N and every descendant of N is deleted. No attribute of any ascendant of N is changed when V is killed.

    1. Execution of a Kill All kills all local variables.
    2. Execution of a Selective Kill kills the variable named in the argument.
    3. In the Exclusive Kill form, <u>name</u> denotes an occurrence of <u>local variable</u> containing no subscripts. All unsubscripted local variables except for the one or more named in the argument are killed. Consequently, all descendants of all local variables are killed except for the descendants of the variables named in the argument.

Cross-reference

Examples

1. KILL X   Local variable X is killed.

2. KILL X,Y,^A(3,4)   Local variables X and Y, and global
                              variable ^A(3,4) are killed.

3. KILL (A)   All local variables except A and
                              its descendants are killed.

4. K (A,C,DEF,%1)   All local variables except those
                              named and their descendants are
                              killed.

5. K   SET X=$T   The argumentless kill causes all
                              local variables to be killed. After
                              execution of this line, X will be
                              the only local variable with a defined
                              value (provided that $T is defined).

6. K:X=1 ^A(X)   If X=1, then ^A(1) is killed. All
                              forms of KILL may have a post-conditional
                              on the command word.

7. K ^A   The entire global array ^A is killed.

8. K @X   Indirection may be used
    K A,@B,C   in the KILL command
    K:@B=C A(B,@C)

LOCK

Syntax

1. Argumentless form:

   LOCK [␣]

2. Form with argument:

   LOCK ␣ | variable name | [timeout]
            | (variable name [, variable name] ...) |

Elaboration

1. The command word may be abbreviated to the single letter L.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The upper argument form, LOCK ␣ variable name [timeout] ,
is equivalent to the lower argument form in which exactly one name
occurs: LOCK ␣ (variable name) [timeout] . Therefore the following
definition of the form with argument deals only with the second,
more general, argument form.

Execution of both forms of LOCK may be viewed in a unified way as
follows. Execution consists of two steps. The first step is common
to both forms. The second step is not executed by the argumentless
form.

Step 1.   The element of the Lock List P-vector corresponding to
          this partition is unconditionally made empty.
Step 2.   Subject to the conditions described below, the element
          of the Lock List P-vector corresponding to this partition
          is given a new value. The following comments apply to
          this process.

In 8.1.4.2 it was stated that the presence of a <u>timeout</u> in a command argument denotes that execution is dependent on an external condition whose definition is associated with the particular command. The condition associated with LOCK is true if and only if the Descendant Exclusivity Rule (referred to in the following as "the Rule", see 4.2) would not be violated by replacement of the now-empty Lock List associated with this partition by a new Lock List denoted by the argument. Stated another way, the Rule mandates a particular kind of noninterference which must hold between all possible pairs of partition Lock Lists. This LOCK argument defines a new Lock List (in a way to be described below) for this partition. The new Lock List, taken as a whole, must not violate the Rule. If indeed it would not, the condition is true; if it would violate the Rule, the condition is false. Of course, the truth of the condition can be a function of time, since other partitions' Lock Lists may be changing.

If the timeout is absent, execution is suspended, if and only if necessary to satisfy the Rule, until assignment of the new value to this partition's Lock List can occur without violation of the Rule. When the Lock List can be given the entire new value, it will, and execution will proceed. The Test Switch will not be altered.

If the timeout is present, the definition of execution is the same as that stated above, with two additional elements. Let t be the nonnegative value implied by the timeout (see 8.1.4.2).

1. If t=0, no suspension of execution occurs. If t is positive, execution is suspended if and only if necessary as specified above, but at the end of t seconds, resumption of execution is forced. (In the following, if t=0, the "time of resumption of execution" is defined to be the time at which execution occurs.)

2. At the time of resumption of execution, the condition is tested. If it is true, the Lock List is given its designated value, the Test Switch is given the value 1, and execution resumes. If the condition is false, the Lock List remains empty, the Test Switch is given the value 0, and execution resumes.

There are no partial assignments to the Lock List of a sublist of the list specified by an argument, and there are no retroactive or delayed changes to a Lock List. If a process fails to establish its Lock List as desired upon execution of a LOCK, any future retries must be explicitly executed by means of the LOCK command.

The Lock List specified by the argument contains as many elements
as the argument contains variable names, and they are in one-to-one
correspondence. For each variable name in the argument, the corres-
ponding Lock List element is the node designator derived from that
variable name. See 12.4 for the syntax of variable name and the
algorithm for deriving its implied node designator.

Since an argument of LOCK is only a name and not a reference to Named
Partition or System Storage, the following propositions hold.
1. There is no relationship between a variable name appearing
   as an argument of LOCK and the existence of, or the attributes
   of, any node in Named Partition or System Storage.
2. The appearance of a global variable as an argument of LOCK
   does not affect the Naked Indicator.

Cross-reference

4.2   Lock List in System Storage
5.3   Test Switch
8.1.4.2   timeout
12.4   Node Designation Mapping, variable name

Examples

1.   LOCK ^A

Between the time execution is resumed
after this LOCK and the time of
execution of the next LOCK by this
process, no other process will be
able to execute an argument of LOCK
containing the unsubscripted global
name ^A or any subscripted global
name whose name part is A. In addition,
any variable names previously LOCKed
by this process are unLOCKed.

2.   L (^B(1,2),^C(3))

Between the time execution is resumed
after this LOCK and the time of
execution of the next LOCK in this
partition, no other process will
be able to LOCK any of the following
variables.
^B   ^B(1)   ^B(1,2,3)
^C   ^C(3)   ^C(3,4,5)
However, another process can LOCK
any of the following.
^B(2)   ^B(1,3,4)
^C(4)

3.	L SET X=1	The argumentless LOCK command releases the effect of any previous LOCK in this partition.
4.	L D(1):3 G A:'$T	An attempt will be made for an interval of three seconds to LOCK D(1). If the attempt is successful, $T will assume a value of 1. If unsuccessful, $T will be zero and execution will continue at the line labeled A.
5.	L D(1):3 ELSE G A	This has the same effect as the example above.
6.	L:X=1 ^EF(^A(5))	The LOCK command can be post-conditionalized. Appearance of a global variable name does not affect the Naked Indicator unless, as in this example, it occurs in an evaluated expression. Here it is a subscript; other such possibilities are in post-conditionals and indirection.
7.	L @A L:@B=C X(@A),@B	Indirection may be used in the LOCK command.

OPEN

Syntax

$$\text{OPEN} \ \_ \ \underline{\text{device specifier}} \ \left[\ : \ \begin{array}{c} \underline{\text{device parameters}} \ [\ \underline{\text{timeout}} \ ] \\ \hline \underline{\text{timeout}} \end{array} \right]$$

$$\underline{\text{device specifier}} \ ::= \ \underline{\text{expression}}$$

$$\underline{\text{device parameters}} \ ::= \ \left| \begin{array}{l} \underline{\text{expression}} \\ (\ \underline{\text{expression}} \ [\ : \ [\ \underline{\text{expression}} \ ] \ ] \ \ldots \ ) \end{array} \right|$$

Elaboration

  1. The command word may be abbreviated to the single letter O.
  2. The command word may be post-conditionalized.
  3. Arguments may be listed.
  4. Arguments may not be post-conditionalized.
  5. Argument indirection is permitted.

Execution

The device specifier in the command argument denotes an input-output "device", which is a sequential character source and/or sink. The value of this expression is to be used as a device designator in the Open List corresponding to this partition, as described in 4.2. The implementor specifies the syntax rules which the value of this expression must satisfy. Also, the implementor specifies the correspondence between the possible values of this expression and the set of available devices.

A partition is said to "own" device D if and only if a device designator designating D is in the partition's Open List.

The intended function of OPEN with one argument is to obtain ownership of one device. Execution of each argument of a multiple-argument form of OPEN will add, in turn, each device designator specified by the argument to the partition's Open List unless doing so would violate the Device Exclusivity Rule ("the Rule"). The Rule states that no two partitions may own the same device. The condition associated with the OPEN command (see 8.1.4.2) is true if and only if adding the specified device designator to the partition's Open List would not violate the Rule.

If there is no timeout in the argument, execution will be suspended
if and only if necessary to avoid violation of the Rule. When the
device designator specified by the argument may be added to the Open
List associated with this partition without violating the Rule, it
will be, and execution will proceed. The Test Switch will not be
affected.

If there is a timeout in the argument, the definition of execution
is the same as that stated above, with two additional elements.
Let t be the nonnegative value implied by the timeout (see 8.1.4.2).

1.   If t=0, no suspension of execution occurs. If t is positive,
      execution is suspended if and only if necessary as specified
      above, but at the end of t seconds, resumption of execution
      is forced. (In the following, if t=0, the "time of resumption
      of execution" is defined to be the time at which execution occurs.)

2.   At the time of resumption of execution, the condition associated
      with the OPEN command is tested. If it is true, the device
      designator is added to the Open List, the Test Switch is given
      the value 1, the device parameters are disposed of as specified
      below, and execution proceeds. If the condition is false,
      the Open List remains unchanged, the Test Switch is given the
      value 0, there is no effect due to the device parameters, and
      execution proceeds.

The device parameters portion of the argument is used to specify
whatever parameters the implementor chooses to associate with the
device in question. The permissible syntax of the value of each
expression in the device parameters, and the interpretation of each
value, is specified by the implementor. This much, however, is common
among all implementations. Device parameters are uniquely stored
in association with each device, not each partition. At system initi-
ation, each device is given an appropriate set of default parameters,
as specified by the implementor. Each time an OPEN, CLOSE, or USE
is executed, one or more device parameters may be changed by the
executed argument. The value of each device parameter persists until
it is replaced by a subsequent successful execution of an argument
of OPEN, CLOSE, or USE in any partition which specifies a value for
the device parameter.

Cross-reference

    4.3   device designator

Examples

1.	OPEN 4	Ownership of device 4 is obtained. If another process already owns device 4, this process will hang until the device is available.
2.	O 4,X,Y	Similarly, devices 4, X, Y are OPENed in turn.
3.	O X::3 G A:'$T	The process attempts to OPEN device X for up to three seconds. If it is successful, $T will assume a value of 1. Otherwise, $T will be zero and execution will continue at the line labeled A.
4.	O X::3 ELSE G A	This is equivalent to the above.
5.	O:X=1 A,B,C	The OPEN command word may be post-conditionalized.
6.	O A:("IO":300:"TTY"),B:("O":800:"MT":20):TIM	In some implementations, additional arguments may be used for setting device-specific parameters.
7.	O @X	Indirection can be used in arguments of the OPEN command.

QUIT

Syntax

QUIT [␣]

Elaboration

1. The command word may be abbreviated to the single letter Q.
2. The command word may be post-conditionalized.

Execution

QUIT may occur in either of two distinct contexts.

1. In the scope of FOR.
2. Not in the scope of FOR.

The function of execution of QUIT not in the scope of FOR is to cause
an exit from the subroutine initiated by a DO or XECUTE argument.
(The initial activation of this process is assumed to be the result
of an externally executed DO.) In terms of the system model, execution
of QUIT not in the scope of FOR pops the top level off the Partition
Stack (see 6.2 paragraph 2). If this leaves the stack empty, HALT
is executed.

Execution of eor, i.e., executing off the end of a routine body,
is equivalent to execution of QUIT not in the scope of FOR.

Execution of QUIT in the scope of FOR immediately terminates execution
of the innermost FOR in whose scope the QUIT occurs. If the line
containing the QUIT has only one FOR, execution of the QUIT terminates
execution of the line. If the line has more than one FOR, execution
of the QUIT causes termination of the innermost FOR and causes ter-
mination of the current execution of the scope of the second-from-
innermost FOR.

Cross-reference

6.2 paragraph 2 Execution of QUIT
8.2.3   DO
8.2.5   FOR
8.2.7   HALT
8.2.19 XECUTE

Examples

1. I X>3 Q   WRITE Y

2. Q:X>3   WRITE Y   These two examples do not have the
                                          same effect because the scope of
                                          the IF extends to the end of the
                                          line. In the first example the
                                          WRITE Y is never executed.

3. Consider the following routine body.

   A F I=1:1:100 D B Q:I+X>6   W I
     W " END" Q
   B S X=I*2 D C W "B"
     Q
   C W "C" Q

   In this routine body:

        a.   The QUIT in line A terminates the FOR loop.
        b.   The QUIT in line A+1 terminates the routine.
        c.   The QUIT in line B+1 ends execution of the DO B causing
             execution to return inside the FOR loop.
        d.   The QUIT in line C ends execution of the DO C causing
             execution to return to line B.

   The output is CB1CB2CB END

READ

Syntax

READ ⌴ | string literal
format
local variable [timeout]
*local variable [timeout]

Elaboration

1. The command word may be abbreviated to the single letter R.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

Execution of READ causes input and/or output on the current device. Each argument form governs the execution of READ according to a different set of rules.

READ string literal causes output of the string literal to the current device, with alteration of the values of CX and CY as described in 5.5. If the current device does not accept output, the effect on CX and CY is still as described in 5.5, but no output operation is performed.

READ format causes output of format control information to the current device, with alteration of the values of CX and CY as described in 8.1.4.1. If the current device does not accept output, the effect on CX and CY is still as described in 8.1.4.1, but no output operation is performed.

READ local variable [timeout] causes an input ASCII data string from the current device to be accumulated until a termination criterion is met. This termination criterion is dependent on whether the timeout is present, as described below.

The implementor defines an "explicit termination procedure" which may be device-dependent. If the explicit termination procedure involves entry of a character, the implementor defines whether or not that character becomes part of the input string when the procedure is executed.

Let t be the nonnegative value associated with the timeout. (See 8.1.4.2.)

1. If the timeout is absent, execution is immediately suspended until the explicit termination procedure is executed; then execution immediately resumes. The input string is the concatenation of all characters received during the time that execution is suspended, with characters received earlier at the left.

2. If the timeout is present and t=0, execution is not suspended and the input string is empty.

3. If the timeout is present and t is positive, execution is immediately suspended. Execution resumes either at the end of t seconds or immediately upon explicit termination, whichever is earlier. The value of the input string is the concatenation of all characters received during the time that execution is suspended, with characters received earlier at the left.

Let I be the value of the input string as defined above. At the time of resumption of execution, the following occurs.

1. Execute SET <u>local variable</u> = I.

2. If the timeout is absent, the Test Switch is unchanged. If the timeout is present and t=0, the Test Switch is given the value 0. If the timeout is present and t is positive and the explicit termination procedure did not occur at or prior to resumption of execution, the Test Switch is given the value 0. If the timeout is present and t is positive and the explicit termination procedure occurred at or prior to the resumption of execution, the Test Switch is given the value 1.

READ *<u>local variable</u> [<u>timeout</u>] is a one-character read, where the input character may be an ASCII character, or it need not be; for example, the input "character" may be a status value.

The local variable named in the argument always receives an integer value. Whether the integer value is associated with a character in a code table (for example, it may be the decimal equivalent of the binary ASCII code of the input character), or whether it has some other interpretation, is a matter which the implementor may define in a device-dependent way.

1. If the timeout is absent, execution is immediately suspended until one input character arrives from the current device; then execution immediately resumes.

2. If the timeout is present and t=0, execution is not suspended and no character is considered to have arrived from the current device.

3. If the timeout is present and t is positive, execution is immediately suspended. Execution resumes either at the end of t seconds or immediately upon receipt of one character from the current device, whichever is earlier.

At the time of resumption of execution, the following occurs.

1. If a timeout is present and either t=0 or t is positive and no character arrived during suspension of execution, execute SET local variable = -1.

2. Otherwise, execute SET local variable = the integer value associated with the input character.

3. If the timeout is absent, the Test Switch is unchanged. If the timeout is present and t=0, the Test Switch is given the value 0. If the timeout is present and t is positive and no input character arrived during suspension of execution, the Test Switch is given the value 0. If the timeout is present and t is positive and an input character arrived during suspension of execution, the Test Switch is given the value 1.

The form READ local variable [timeout] (that is, the form "without asterisk") affects CX and CY as described in 5.5. The form with asterisk may also affect CX and CY according to 5.5, depending on the implementor's interpretation of the input character.

A strict reading of the execution definition of the last two forms of READ leads to the conclusion that any character or explicit termination procedure arriving prior to the execution of this READ and, if there was a prior READ, after its execution, will be lost. This interpretation may be undesirable in certain operating environments. Therefore, the implementor has latitude to make some or all of the following re-interpretations, in a consistent, but possibly device-dependent, way.

1. Input characters are queued and received from the current device
   by the READ command in first-in, first-out order in such a way
   that no character is lost. In the case of READ without asterisk,
   all characters arriving from the current device prior to execution
   of this READ and after all prior READs are concatenated to the
   left of the input string described above. In the case of READ
   with asterisk, the first character to arrive after all prior
   READs is the one processed.

2. In the case of READ without asterisk, the implementor must
   decide how to treat multiple explicit terminations. There are
   two choices.
   a. Terminations are queued with, and partition, input characters,
      and each READ accepts only one such partition.
   b. If, at the start of the READ, a character has been received
      since the most recent termination, all prior terminations
      are ignored. If a prior termination has not been followed
      by a character, then that termination is considered to
      occur immediately after the start of the READ.

3. When a timeout is present, the case t=0 ceases to be a special
   case and may, along with positive values of t, give a value
   of 1 to the Test Switch if the condition being tested (explicit
   termination or arrival of a character, respectively) occurred
   prior to this READ and subsequent to all prior READs. However,
   assignment to the Test Switch of the value 1 means, and only
   means, that the value given to the named local variable is as-
   sociated with a specific condition. In the case of READ without
   asterisk, the explicit termination procedure was executed.
   In the case of READ with asterisk, the value assigned to the
   local variable is associated with an input character and is
   not the result of the default assignment of the value -1 to
   the variable.

Cross-reference

Examples
===

1. READ X   A data string is entered into local
                                     variable X.

2. R X,Y,Z

3. R "PATIENT?",X   The message PATIENT? appears on
                                     the current device; then execution
                                     waits for data to be entered, after
                                     which it is placed into X.

4. R !!,"FIRST?",X,?30,"SECOND?",Y
                                     This code causes two carriage return-
                                     line feeds to occur on the current
                                     device. Then the message FIRST?
                                     is displayed. After data is entered
                                     into variable X, the device spaces
                                     to column 30 and the message SECOND?
                                     is displayed. Data may then be
                                     entered and read into variable Y.

5. R "TIMING?",X:10 G A:'$T
    R "TIMING?",X:10 E G A
                                     These two lines do the same thing.
                                     After displaying TIMING? the job
                                     will hang for ten seconds waiting
                                     for input. If the return key (the
                                     assumed termination procedure) is
                                     pushed during that time period,
                                     $T will be set to 1. Otherwise,
                                     $T will be 0, X may contain partial
                                     input, and execution will continue
                                     at A. If no characters have been
                                     entered, X will contain the empty
                                     string.

6. R *X,*Y,*Z   Under one interpretation available
                                       to the implementor, this form of
                                       READ may be used for entry of single
                                       characters and conversion to their
                                       ASCII-equivalent numeric codes.
                                       If the characters AB CR were entered,
                                       the following values would result.
                                       X=65
                                       Y=66
                                       Z=13

7. R *X:10 E G A   The asterisk form may also use a
                                       timeout. If the timeout expires
                                       without a character having been
                                       entered, X will have the value -1
                                       and execution will continue at A.

8. R:$D(^A) !!,"TEST",X:12,*Y,*Z:10,#,"SAMPLE",!,A:9
                                       The READ command may be post-conditional-
                                       ized. All of the various argument
                                       forms may appear in one argument
                                       list. When more than one timeout
                                       is present, the value of $T reflects
                                       the last-executed (rightmost) timeout.

9. R @X   Indirection can be used with READ.
    R X:@Y
    R:@C @X:@Y
    R *@A

SET

Syntax

$$\text{SET } \underset{\sqcup}{} \quad \left|\begin{array}{l} \underline{\text{storage reference}} \\[6pt] (\underline{\text{storage reference}} \text{ [,}\underline{\text{storage reference}}] \text{ ...)} \end{array}\right| \quad = \underline{\text{expression}}$$

Elaboration

1. The command word may be abbreviated to the single letter S.
2. The command word may be post-conditionalized.
3. Arguments may be listed. (The form shown above is a single argument.)
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The SET command provides the means whereby a data string is placed into the value attribute of one (or more) node(s) of Named Partition or System Storage. The data string is the value of the expression; the node(s) is (are) that (those) designated by the storage reference(s). (The words in parentheses apply to the parenthesized form of the argument.)

Execution occurs in the following order.

1. Any indirections or subscripts in the storage reference(s) are evaluated, in left-to-right order as they appear in the argument.
2. The expression is evaluated. Let V denote the value of the expression.
3. For each storage reference (one at a time in left-to-right order if there are several), V is placed into the Value attribute of the node of Named Partition Storage or Named System Storage denoted by the storage reference. (The details of this mapping from a storage reference to a node designator are discussed in 12.4.) This may involve the following considerations.

   a. If the storage reference is a naked reference, the content of the Naked Indicator is used to produce the node designator, and then this resulting node designator causes the content of the Naked Indicator to be given a (possibly new) value. The process is described in 5.2 and 12.4.

b. If there does not exist a node with a Name attribute
    as denoted by the storage reference, a minimum number
    of nodes is created so that
    (1) A node with the Name attribute defined by
         the storage reference exists, and
    (2) It is a descendant of the directory node
         in either Named System Storage (if the first
         character of the Name attribute is "^")
         or Named Partition Storage (otherwise).
c. If any nodes are so created, their Name attributes
    are defined by the description of the creation process
    given above. Their other attributes are defined as
    follows.
    (1) The node whose Name attribute is defined
         by the storage reference is given the D
         attribute 1 and the Value attribute V.
    (2) Any other nodes so created (and they form
         an unbroken chain of ascendants of the above
         node beginning with its immediate ascendant)
         are given the D attribute 10. Their value
         attributes, being unavailable, need not
         be given any value.
d. Also, if any node is so created, there exists a unique
    node which, prior to the creation had no descendant
    and after the creation has all of the created nodes
    as descendants. The D attribute of this node is given
    the following value.

If it was	then	it becomes
0		10
1		11
10		10
11		11

e. If the node whose Name attribute denoted by the storage
    reference does exist, V is placed into its Value attribute
    and its D attribute is given the following value.

If it was	then	it becomes
0		1
1		1
10		11
11		11

Cross-reference

Examples

1.   SET ^X=Y

        The content of the Value attribute of the level 1 Partition node named Y (i.e., the "value of the unsubscripted local variable Y") is placed into the Value attribute of the level 1 System node (i.e., the "unsubscripted global variable") named ^X.

2.   S ^A(1,2)=B(3,4)

        Similarly for subscripted variables.

3.   S ^A(1,2)=X,B(4)=7,^C="TEST",X=^D

        Multiple arguments of SET.

4.   S ^(2)=^C(X,2)+1

        Because the expression to the right of the = is evaluated before the variable name to its left is determined, the naked reference ^(2) denotes ^C(X,2). If the line had been
           S ^C(X,2)=^(2)+1
the meaning of the naked reference would depend on the state of the Naked Indicator prior to execution of the SET.

5.   S (A,B,C)=X

        This is the "multiple" set. All variables in the parentheses are set to the same value.

6.   S (A,B,C,D)=0,^A(3)=W,(^A(3,4),^B(5))=I

        Multiple and single SETs may be mixed in one command.

7.   S ^A(5)=10,B(^(3))=^C(4)

        When subscripts occur to the left of the =, they are evaluated before the right-side expression. Thus this line is equivalent to
           S ^A(5)=10,B(^A(3))=^C(4) .

8. S ^A(3)=10,^(^(3))=^C(4)

This line is executed as follows.
a.   ^A(3) is given the value 10.
b.   The subscript ^(3), which is ^A(3), is evaluated as 10.
c.   ^C(4) is evaluated.
d.   This value is given to ^(10), which is ^C(10).
This line is therefore the same as
    S ^A(3)=10,^C(^A(3))=^C(4)   .
The order is always:
a.   First the left-hand subscripts.
b.   Then the right-hand expression.
c.   Then the left-hand name.

9. S I=3,(I,A(I))=4

Even in a multiple SET, the left-hand subscripts are evaluated before the assignment is made. But each argument is completed before the next is begun. Thus, this line is equivalent to
    S I=3,(I,A(3))=4
and I ends up with the value 4.

10. S:X=3 X=4.5

The SET command may be post-conditionalized. In this example, if X has the value 3, then it is given the value 4.5.

11. S X=$S(X=3:4.5,1:X)

The $SELECT function is valuable when used in the right-hand expression of a SET command. This line has the same effect as the previous example.

12. S X=X=3*1.5+X

The expression to be evaluated may be quite complex. In this example, the expression X=3*1.5+X is evaluated, and the result is given to the variable X. This line has the same effect as the previous two examples.

13. S @X
     S @A=B+C
     S A=@B+C
     S A=B+@(C_D)

Indirection is often used in the SET command.

USE

<u>Syntax</u>

        USE ⎵ <u>device specifier</u> [: <u>device parameters</u>]

      <u>device specifier</u> ::= <u>expression</u>

      <u>device parameters</u> ::= | <u>expression</u> |
                                 | (<u>expression</u> [: [<u>expression</u>]] ...) |

<u>Elaboration</u>

    1.   The command word may be abbreviated to the single letter U.
    2.   The command word may be post-conditionalized.
    3.   Arguments may be listed.
    4.   Arguments may not be post-conditionalized.
    5.   Argument indirection is permitted.

<u>Execution</u>

The device specifier in the command argument denotes an input/output "device", which is a sequential character source and/or sink. The value of this expression is to be used as a device designator in the partition's Current Device Designator, as described in 5.4.

The intended function of USE is to make the designated device the current device by moving the value of the device specifier into the Current Device Designator. Execution proceeds as follows.

1.   If, and only if, the content of the Current Device Designator designates a device, the values of CX and CY are placed into the X and Y registers specific to that device.

2.   One of the following two alternatives exists.

        a.   If the value of the device specifier designates a device <u>and</u> if there is a device designator in the partition's Open List which designates the same device, then the value of the device specifier is placed into the Current Device Designator, the X and Y values specific to that device are placed into CX and CY, respectively, and the device parameters are recorded as described under OPEN (8.2.12). (Translation: The device designated by the USE command is made the current device of the process executing it.)

        b.   Otherwise, an erroneous condition exists.

The value of the Current Device Designator may be obtained by execution of the special variable $IO as an expression atom.

Cross-reference

    4.3   device designator
    5.4   Current Device Designator
    8.2.12   device parameters
    Chapter 9 $IO special variable

Examples

1.	USE 3	Device 3 is specified for routing of READ and WRITE data.
2.	U 3 SET X=$I	X will receive the value 3.
3.	U X:Y	In some implementations, Y may be used to specify device parameters.
4.	U:X'=0 X	The USE command may be post-conditionalized. This example inhibits making device 0 the current device.
5.	U @X	Argument indirection is permitted.

VIEW

<u>Syntax</u>

      VIEW argument syntax specified by implementor

<u>Elaboration</u>

View is an implementation-specific command made available to implementors who may wish to output data not otherwise available. Whether or not it takes one or more arguments, and what any argument syntax is, are specified by the implementor. Each implementation must recognize and accept the VIEW command regardless of any interpretation given to it.

[Port: Routines designed for portability should not contain the VIEW command. :Port]

<u>Syntax</u>

$$\text{WRITE} \ \sqcup \ \left| \begin{array}{l} \underline{\text{format}} \\ \underline{\text{expression}} \\ \underline{*\text{integer expression}} \end{array} \right|$$

<u>Elaboration</u>

1. The command word may be abbreviated to the single letter W.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

<u>Execution</u>

Execution of an argument of WRITE causes output of data and/or control information to the current device. Each argument form governs the execution of WRITE according to a different set of rules.

WRITE <u>format</u> causes output of format control information to the current device, with alteration of the values of CX and CY as described in 5.5 and 8.1.4.1. If the current device does not accept output, the effect on CX and CY is still as described there, but no output operation is performed.

WRITE <u>expression</u> causes output of the value of the expression, one character at a time, in left-to-right order. The effect of each character at the device is defined by the ASCII standard and conventions. Each character of the output affects the values of CX and CY as described in 5.5. If the current device does not accept output, the effect on CX and CY is still as described in 5.5, but no output operation is performed.

WRITE *integer expression is a one-character write whose possibly device-dependent interpretation is defined by the implementor. Possible interpretations include the following.

1. A device command, such as a tape-unit rewind.
2. A numeric device parameter, such as a disk arm position or an absolute plotter coordinate.
3. Any ASCII character the decimal equivalent of whose binary code is the value of the integer expression.

The implementor defines the effect of this form of WRITE on CX and CY.

It is intended that, except for the effects of timeouts, READ and WRITE have identical side effects for identical data transfers.

Cross-reference

5.4 Current device
5.5 CX and CY
8.1.4.1 format
8.2.14 READ

Examples

1. WRITE X

2. W !,"Line feed",#,"Form feed",?20,"TAB",A(3)

   The arguments of the WRITE command may be a mixed selection.

3. W *7,*7,*97

   Under interpretation 3 above, this command would ring the bell twice, then output an "a".

4. W X,*Y,*Z,!!,"TEST",A=B

   Expressions containing operators may be used as arguments.

5. W:X>10 $J(X,7,2)

   The WRITE command may be post-conditionalized. The $JUSTIFY function is frequently used in arguments of WRITE.

6. W @X
   W Z,@X,Y

   Indirection may be used in arguments of WRITE.

Syntax

        XECUTE ⌴ <u>expression</u>

Elaboration

    1.   The command word may be abbreviated to the single letter X.
    2.   The command word may be post-conditionalized.
    3.   Arguments may be listed.
    4.   Arguments may be post-conditionalized.
    5.   Argument indirection is permitted.

Execution

    Execution of an argument of XECUTE is described in full in terms
    of the system model in 6.2 paragraph 6. Its effect is to call a
    one-line subroutine whose spelling is the value of the <u>expression</u>,
    preceded by <u>ls</u> and followed by <u>eol</u>. Control automatically returns
    to the argument or command following the XECUTE argument upon completion
    of or exit from the subroutine. The implied routine-body context
    of this subroutine (necessary for interpretation of a $TEXT or local
    <u>entry reference</u> in the subroutine) is the routine body containing
    the XECUTE.

Cross-reference

    6.2 paragraph 6   Execution of XECUTE
    8.2.13   QUIT

Examples

    1.   XECUTE A   If A has the value "S X=1", then
                                     X will receive the value 1.

    2.   X "S X=B Q:X<3 W Y" W "OUT"
                                     In this example the QUIT in the
                                     value of the argument may terminate
                                     the execution of the subroutine.
                                     The "OUT" is always written.

    3.   X "G A" W "OUT"   The "OUT" is never written.

    4.   X A,B_" D 3 G "_C,D   XECUTE may have multiple arguments.
                                     Note also the use of concatenation
                                     for stringing together commands
                                     and arguments.

    5.   X:Y=1 A_" X B,C",Y   XECUTE with a post-conditional.
                                     Note the nesting of XECUTEs.

    6.   X @Y,"I @A G "_@D   An example with indirection at two
                                     levels.

Syntax

>Z [spelling of remainder of command word is specified by the
implementor]   argument syntax is specified by the implementor

Elaboration

>Implementors desiring to offer commands not in the standard are required
by the standard to spell the command words beginning with Z.

>[Port: Routines designed for portability should not contain Z commands.
:Port]

CHAPTER 9

SPECIAL VARIABLES

Table of Contents

CHAPTER 9

SPECIAL VARIABLES

A special variable is an upper-case alphabetic name from a prescribed list, preceded by a $, which, when executed as an expression atom, yields a value which is in System Storage or Partition Storage but is not otherwise explicitly available. The following special variables are defined.

$HOROLOG	The content of the Clock Register.
$IO	The content of the Current Device Designator.
$JOB	The content of the element of the Job Number P-vector corresponding to this partition.
$STORAGE	A measure of space available for the routine and Named Partition Storage.
$TEST	The content of the Test Switch.
$X	The content of CX.
$Y	The content of CY.

$Z[implementor-defined]: as defined by the implementor.

Special variables may be abbreviated to two characters: the $ followed by the first letter of the name.

The syntactic entity special variable is defined as follows.

```
                                    | $H[OROLOG]
                                    | $I[O]
                                    | $J[OB]
      special variable    ::=       | $S[TORAGE]
                                    | $T[EST]
                                    | $X
                                    | $Y
                                    | $Zimplementor-specified
```

$HOROLOG

Name: $HOROLOG

Abbreviation: $H

Value: The content of the Clock Register at the time of execution.

Cross-reference: 4.5 Clock Register

Examples:

1. The following code converts $H to a readable time format such as 11:15:01 AM

   ```
   S TIME=$P($H," ",2),SECONDS=TIME#60
   S X=TIME\60,HOURS=X\60,MINUTES=X#60
   S SUFFIX=$P("AM,PM",",",HOUR\12+1)
   W !,HOUR#12+1,":",MINUTES,":",SECONDS," ",SUFFIX
   ```

2. The following code converts $H to a date in the form MM/DD/YYYY.

   ```
   S H=$H>21607+$H,LEAPYRS=H\1461,Y=H#1461
   S YEAR=LEAPYRS*4+1841+(Y\365),DAY=Y#365+1
   S MO=1 I Y=1460 S DAY=366,YEAR=YEAR=1
   F I=31,YEAR#4=0+28,31,30,31,30,31,31,30,31,30,31 Q:DAY'>I  S DAY=DAY-I,MO=MO+1
   W MO\10,MO#10,"/",DAY\10,DAY#10,"/",YEAR
   ```

Name: $IO

Abbreviation: $I

Value: The content of the Current Device Designator.

Cross-reference: 5.4 Current Device Designator

Examples:

1. USE 3 S X=$I   X is given the value 3.

2. S ^A($I,1)=X   Scratch files can be set up with
                                    data stored by device number.

$JOB

Name: $JOB

Abbreviation: $J

Value: The content of that element of the Job Number P-vector
                   corresponding to this partition.

Cross-reference: 4.4 Job Number P-vector

Examples:

1.   S ^A($J,2)=Y   Scratch files can be set up with
                                     data stored by job number.

2.   WRITE #,DATE," ",$J   A way of identifying the source
                                     of output.

<u>Name</u>:	$STORAGE
<u>Abbreviation</u>:	$S
<u>Value</u>:	The value of $S is not precisely defined by the Standard because it is necessarily somewhat implementation-dependent. $S measures, at the time of its execution, the number of additional characters which may be added to the "partition space" before the system refuses to allow further expansion of the partition space. In the absence of a more precise formula provided by the implementor, the Portability Requirements suggests the following model. Partition space PS is computed as the following sum.

$$PS = RS + LVS + TRS$$

RS (routine size) is the number of characters in all lines of the routine body at the top of the Routine Stack (where <u>ls</u> counts as one character and <u>eol</u> counts as two characters).

LVS (local variable storage) is the total number of characters in all Value attributes of all nodes of Named Partition Storage (undefined values are considered empty) plus the total number of <u>data</u> characters (¢ is excluded) in all Name attributes of such nodes plus two times the number of such nodes plus two times the number of pairs of such nodes which are in the immediate ascendant/immediate descendant relationship.

TRS (temporary result storage) is the number of characters of temporary data values, arising from execution of the current line, in existence at the time of execution of $S. These include
1. Characters which have been added to and remain in the line buffer as a result of indirection.
2. Intermediate results in the evaluation of the expression containing the $S due to incomplete parenthesized right-hand operands of binary operators.
3. If the $S occurs in a SET, the data characters in the <u>storage reference</u> node designators which are both to the left of the = and to the left of the $S.

[Port: PS is not to exceed 4000 at any time during execution of a routine designed for portability. :Port]

If TAS is the "total available space" measured in characters at the time of execution of the $S,

$$\$S = TAS - PS \ .$$

Note, however, that TAS may be a function of activity outside the partition in which the $S is executed.

Cross-reference: 5.1 Named Partition Storage (See also 4.1)
6.2 paragraph 8 Indirection
Chapter 7 Structure of lines
8.2.15 left-hand side of SET

Examples:

1.   I $S<50 G ^OVERFLOW   A test for $S below a threshold permits preparing for a storage overflow.

2.   F I=1:1:100 DO 3 I $S<100 D ^FILE

Name: $TEST

Abbreviation: $T

Value: The content of the Test Switch (0 or 1). The Test
                        Switch must be defined at the time of execution of
                        $T, ELSE, or argumentless IF.

Cross-reference: 5.3 Test Switch
                        8.1.4.2 timeouts
                        8.2.4 ELSE
                        8.2.9 IF

Examples

1.   I X=3 S Y=A   This code is equivalent to
      W:$T Y S Y=B   I X=3 S Y=A
                                      I   W Y
                                      S Y=B

2.   R X:100 G A:'$T DO B   In this example, $T is used to test
                                      whether the input was explicitly
                                      terminated before the timeout ex-
                                      pired. If there was no termination
                                      (i.e., if the input string is incomplete),
                                      execution continues at A.

3.   O 4:10 G A:'$T U 4 W X   This is similar to example 3; execution
                                      transfers to A if the OPEN is unsuccessful.

$X

Name: $X

Abbreviation: $X

Value: The content of CX

Cross-reference: 5.5   CX

Examples:

1.   W !,"123" S Y=$X   Y is given the value 3.

2.   W ! S Y=$X   Y is given the value 0.

3.   W:$X>72 !   This code writes a carriage return,
                                  line feed if $X indicates that the
                                  next output character would be to
                                  the right of column 72.

Name: $Y

Abbreviation: $Y

Value: The content of CY

Cross-reference: 5.5   CY

Examples:

1.   W # S Y=$Y   Y is given the value 0.

2.   W #!!! S Y=$Y   Y is given the value 3.

3.   W:$Y>56 #   If the page is full, skip to a new
                                          page.

$Z

Name: $Zspecified by implementor

Abbreviation: $Zspecified by implementor

Value: Specified by implementor. The Standard requires that
                         all implementor-defined special variables begin with
                         $Z.

                         [Port: Routines designed for portability should not
                         contain $Z special variables. :Port]

CHAPTER 10

FUNCTIONS

Table of Contents

CHAPTER 10

FUNCTIONS

10.1 General Information

Each MUMPS function has one or more function arguments and, when it is executed as an expression atom, yields a value which is a data string. While the specific syntax of each function is separately prescribed, all functions have the following general syntax.

1. A function is a function name followed by an argument list enclosed in parentheses.
2. A function name is a "$" followed by an upper-case alphabetic name from the prescribed list of function names.
3. In the case of multiple function arguments, adjacent arguments are separated by a comma.

The following function names are defined.

$ASCII   selects a character of a string and returns its code as an integer.

$CHAR   translates a list of integers into a string of characters whose ASCII codes are those integers.

$DATA   returns the D attribute of the designated node of Named System or Named Partition Storage.

$EXTRACT   returns a character or substring of a string expression, selected by position number.

$FIND   returns an integer specifying the end position of a specified substring within a string.

$JUSTIFY   returns the value of an expression, right-justified in spaces within a field of specified size.

$LENGTH   returns the length of a string.

$NEXT   identifies the next (in order of ascending last integer subscript) sibling node of a specified node of Named System or Named Partition Storage.

$PIECE   returns the string between two specified occurrences of a specified substring within a specified string.

$RANDOM   returns a pseudo-random number in a specified interval.

$SELECT   returns the value of one of several expressions in a list, selected by the truth values in a second list of expressions.

$TEXT   returns the text content of a specified line of the routine body at the top of the Routine Stack.

$VIEW   an implementor-defined function available for providing implementation-specific data.

$Z[implementor-defined]
        as defined by the implementor.

Function names may be abbreviated to two characters: the $ followed by the first letter of the name.

The syntactic entity function is defined as follows.

```
                 $A[SCII](expression[,integer expression])
                 $C[HAR](integer expression[,integer expression]...)
                 $D[ATA](storage reference)
                 $E[XTRACT](expression,integer expression[,integer expression])
                 $F[IND](expression,expression[,integer expression])
                 $J[USTIFY](expression,integer expression)
                 $J[USTIFY](numeric expression,integer expression,integer expression)
function ::=     $L[ENGTH](expression)
                 $N[EXT](storage reference)
                 $P[IECE](expression,expression,integer expression[,integer expression])
                 $R[ANDOM](integer expression)
                 $S[ELECT](truth-value expression:expression
                           [,truth-value expression:expression]...)
                 $T[EXT](+integer expression)
                 $T[EXT](line reference)
                 [$V[IEW](argument(s) specified by implementor)]
                 [$Zname specified by implementor(argument(s) specified by
                           implementor)]
```

10.1.1 Definition of Position Number

If a nonempty string S has n characters, each character is given a unique Position Number in the closed interval [1,n]. The leftmost character of S has the position number 1, the rightmost character of S has the position number n, and intermediate character positions map into the intermediate integers in the expected way.

The reverse mapping $E(S,p)$ from an integer-valued position number p in S to a string containing zero or one character is defined as follows.

1. If the position number p is less than 1 or greater than n, the value of $E(S,p)$ is the empty string. (This is also the case for all values of p whenever S is empty.)
2. If the position number p is in the interval [1,n], the value of $E(S,p)$ is the one-character string whose character is the character of S with position number p.

This reverse mapping is the definition of the two-argument form of $EXTRACT(S,p)$.

$ASCII

Name: $ASCII

Abbreviation: $A

Syntax: $A[SCII](<u>expression 1</u> [,<u>integer expression 2</u>])

Value: If <u>integer expression 2</u> is absent, a default value
                   of 1 is used.

                   $ASCII yields an integer value which is the decimal
                   equivalent of the ASCII code of the value of
                   $E(<u>expression 1,integer expression 2</u>). If that value
                   is empty, the value of $ASCII is -1.

Cross-reference: 10.1 $E
                   Appendix A   ASCII code table

Examples:

Value of first argument string	Value of function
SET X="ABCDE"	$A(X)=65
	$A(X,1)=65
	$A(X,2)=66
	$A(X,3)=67
SET Y=4	$A(X,Y)=68
SET X="" (the empty string)	$A(X)=-1
	$A(X,n)=-1 for all n
SET X="AB"	$A(X,0)=-1
	$A(X,3)=-1
	$A(X,-7)=-1
	$A(X,1.92)=65 (that is, the integer value is used)

\$CHAR

Name: \$CHAR

Abbreviation: \$C

Syntax: \$C[HAR](integer expression [,integer expression] ...)

Value: \$CHAR returns a string whose length is the number
                       of argument expressions which have integer values
                       in the closed interval [0,127]. Each integer expression
                       whose value is in that interval maps into the ASCII
                       character whose code (expressed as a decimal number)
                       is the integer expression's value; this mapping is
                       order-preserving. Each negative-valued argument maps
                       into no character in the value of \$CHAR. Values above
                       127 are considered to be erroneous.

Cross-reference: Appendix A   ASCII code table

Examples:

Value of arguments   Value of function

SET X=65,Y=66,Z="GOB"   \$C(X)="A"
                                           \$C(Y)="B"
                                           \$C(X,Y)="AB"
                                           \$C(X,Y,67)="ABC"
                                           \$C(X,-1,Y)="AB"
                                           \$C(-1)="" (the empty string)
                                           \$C(0)=the ASCII NUL character
                                           \$C(\$A(Z,1),\$A(Z,2),\$A(Z,3))="GOB"

$DATA

Name: $DATA

Abbreviation: $D

Syntax: $D[ATA](storage reference)

Value: If the node designated by the argument exists, the
                  value of the function is the content of its D attribute.
                  If the node does not exist, the value of the function
                  is 0. If the storage reference is a naked reference,
                  the Naked Indicator must be defined. If the storage
                  reference is a global variable or a naked reference,
                  the Naked Indicator's content may change.

Cross-reference: 4.1 D attribute
                  5.2 Naked Indicator
                  12.4 storage reference

Examples:

1.   Assume that Named Partition Storage is in its initial state.

Argument values   Function values

Y has no defined value $DATA(Y)=0
SET Y=100 $DATA(Y)=1
SET Y="AB" $D(Y)=1
SET A(1)="TEST" $D(A(1))=1
                                       $D(A)=10
SET B(1,2)="SAMPLE" $D(B(1,2))=1
                                       $D(B(1))=10
SET B(1)="ANOTHER SAMPLE" $D(B(1))=11
KILL B(1,2) $D(B(1,2))=0
                                       $D(B(1))=11   (the KILL does not
                                       change the D attribute of any ascendant)

2. Assume that the argument is a global variable and that Named System
    Storage is in its initial state except for the result of executing
    SET ^A(1,2,3)="TEST".

Action	Value of X	Resulting value of Naked Indicator
SET X=$D(^A)	10	undefined
SET X=$D(^(1))	error	undefined
SET X=$D(^A(1))	10	^A
SET X=$D(^A(99))	0	unchanged
SET X=$D(^(1))	10	unchanged
SET X=$D(^(1,2))	10	^A ¢ 1
SET X=$D(^(2))	10	unchanged
SET X=$D(^(2,3))	1	^A ¢ 1 ¢ 2
SET X=$D(^(3))	1	unchanged
SET X=$D(^(4))	0	unchanged
SET X=$D(^A(1,2))	10	^A ¢ 1
SET ^A(1,2,3,4)="H",X=$D(^A(1,2,3))		
	11	^A ¢ 1 ¢ 2

$EXTRACT

Name: $EXTRACT

Abbreviation: $E

Syntax:

$E[XTRACT](expression 1,integer expression 2 [,integer expression 3])

Value: If integer expression 3 is absent, the definition
                    given in 10.1.1 is used. If integer expression 3
                    is present, the value of $EXTRACT is the contiguous
                    substring of expression 1 defined by the following
                    concatenation formula.

                    If the value of integer expression 3 is not less than
                    the value of integer expression 2,

                    $E(S,m,n) = $E(S,m)_$E(S,m+1)_ ... _$E(S,n-1)_$E(S,n) .

                    If the value of integer expression 3 is less than
                    the value of integer expression 2, the value of the
                    function is the empty string.

Cross-reference: 10.1.1 Definition of two-argument form of $E

Examples:

Prior condition	Function value
SET X="ABCDE"	$E(X,1)="A"
	$E(X,2)="B"
	$E(X,1,2)="AB"
	$E(X,1,4)="ABCD"
	$E(X,0,100)="ABCDE"
	$E(X,1.9)="A" (the integer value of the argument is used)
	$E(X,99)="" (the empty string)
	$E(X,-3)="" (the empty string)
	$E(X,3,2)="" (the empty string)
SET A(1)="2BC"	$E(A(1),A(1))="B" (the integer interpretation of the second argument is used)

10.2.5

Name:	$FIND
Abbreviation:	$F
Syntax:	$F[IND](expression 1,expression 2[,integer expression 3])

Value:

If integer expression 3 is absent, or if its value
is less than 1, a default value of 1 is used. $FIND
searches for an instance of the value of expression 2
as a substring of expression 1. The search is conducted
from left to right beginning at the character whose
position number is the value of integer expression 3.

If the value of integer expression 3 is greater than
the length of expression 1, or if no instance of
expression 2 is found, $FIND returns zero. Otherwise,
the first (leftmost) instance found is the one reported,
as follows. If the position numbers in expression 1
of the found instance of expression 2 are the integers
in the interval [i,j], the value of $FIND is j+1.

Cross-reference: 10.1.1 Position Number

Examples:

Prior condition	Function value
SET X="ABCAX"	$F(X,"A")=2
	$F(X,"B")=3
	$F(X,"Z")=0
	$F(X,"ABC")=4
	$F("ABC","ABC")=4
	$F(X,"A",1)=2
	$F(X,"A",2)=5
	$F(X,"A",4)=5
	$F(X,"A",5)=0
	$F(X,"A",100)=0
	$F(X,"")=1
	$F(X,"",4)=4
	$F(X,"",10)=0
SET Y="B"	$F(X,Y)=3
SET Z="1.2W"	$F(X,Y,Z)=3

$JUSTIFY

Name: $JUSTIFY

Abbreviation: $J

Syntax:

$J[USTIFY](| expression 1,integer expression 2 |)
| numeric expression 1,integer expression 2,integer expression 3 |

Value: The two-argument form right-justifies expression 1 in a field of integer expression 2 spaces. Let Ll be the length of expression 1 and let N2 be the value of integer expression 2. If N2 < Ll there is no truncation; the value of $J is the value of expression 1. If N2 > Ll the value of $J is N2-Ll spaces on the left concatenated with the value of expression 1 on the right.

In the three-argument form, also let N3 be the value of integer expression 3. This form edits the value of numeric expression 1, right-justified, in a field of N2 spaces with N3 decimal places. Specifically, let R be the value of numeric expression 1 after rounding to N3 fraction digits, including possible trailing zeros. (If N3=0, R contains no decimal point.) The value returned by $J is $J(R,N2).

[Port: Negative values of N3 are reserved for future expansion of the definition of $J, and therefore should be avoided by implementors as well as users. :Port]

Cross-reference:

Examples:

Prior condition	Function value
SET X=12.35	$J(X,6)=" 12.35"
	$J(X,5)="12.35"
	$J(X,4)="12.35"
	$J(X,3)="12.35"
	$J(X,7,4)="12.3500"
	$J(X,7,3)=" 12.350"
	$J(X,7,2)=" 12.35"
	$J(X,7,1)=" 12.4"
	$J(X,7,0)=" 12"
SET Y=197	$J(Y,7,2)=" 197.00"
SET Z=5.4	$J(Z,7,2)=" 5.40"
SET W=1.487	$J(W,7,2)=" 1.49"
	$J(W,7,1)=" 1.5"
	$J(W,7,0)=" 1"

Name: $LENGTH

Abbreviation: $L

Syntax: $L[ENGTH](expression)

Value: $LENGTH(expression) returns the integer number of
                         characters in the value of expression. If the value
                         of expression is empty, $L returns zero.

Cross-reference:

Examples:

Prior condition   Function value

SET X="ABC" $L(X)=3
SET X="123456789" $L(X)=9
SET X="" (the empty string) $L(X)=0

$NEXT

Name:	$NEXT
Abbreviation:	$N
Syntax:	$N[EXT] (storage reference)

Value:

Only subscripted (level 2 or greater) forms of the argument are permitted, and the value of the rightmost subscript must be an integer whose value is -1 or greater. If the argument is a naked reference, the Naked Indicator must be defined.

Let the argument be of the form

$$N(s\underline{1},s\underline{2}, \ldots, s\underline{n-1},s\underline{n})$$

where sn is constrained as stated. $NEXT reports the existence and identity of a storage node denoted by the storage reference

$$N(s\underline{1},s\underline{2}, \ldots, s\underline{n-1},t) \quad, \text{ such that}$$

1. the node has a nonzero D attribute;
2. t is an integer greater than sn; and
3. t is the smallest possible value meeting these conditions.

If such a node exists, the value of $NEXT is t; if no such node exists, the value of $NEXT is -1.

Cross-reference:

4.1 Levels of storage
4.1 D attribute
12.4 storage reference

Examples:

Assume that Named Partition and System Storage are in their initial states except for the results of execution of SET ^A(1,2,3)="TEST",^A(4)="SAMPLE".

Action	Value of X	Resulting value of Naked Indicator
SET X=$N(^A)	error	undefined
SET X=$N(^A(-1))	1	^A
SET X=$N(X(4))	-1	unchanged
SET X=$N(^(1))	4	unchanged
SET X=$N(^(4))	-1	unchanged
SET X=$N(^(4,3))	-1	undefined
SET X=$N(^A(1,-1))	2	^A ¢ 1
SET X=$N(^(2,-1))	3	^A ¢ 1 ¢ 2
SET X=^(3)	"TEST"	unchanged
SET X=$N(^(3))	-1	unchanged
SET X=$N(^B(1,2))	-1	undefined

Name: $PIECE

Abbreviation: $P

Syntax:

$P[IECE](expression 1,expression 2,integer expression 3 [,integer expression 4])

Value: If integer expression 4 is absent, a default value
                    which is the value of integer expression 3 is used.
                    Let S1 denote the value of expression 1, let S2 denote
                    the value of expression 2, let N3 denote the value
                    of integer expression 3, and let N4 denote the value
                    of integer expression 4.

                    $PIECE(S1,S2,N3,N4) searches for, and returns as its
                    value, a substring of S1 whose left and right boundaries
                    are delimited by values specified by S2, N3, and N4.
                    Typically, the substring is bounded on the left by
                    (but does not include) the N3-1th occurrence of S2
                    in S1, counting from the left end of S1, and the sub-
                    string is bounded on the right by (but does not include)
                    the N4th occurrence of S2 in S1, counting from the
                    left end of S1. Since empty substrings may be found
                    anywhere in any number required, $P returns an empty
                    string if S2 is empty.

                    Any of the following conditions causes $PIECE to return
                    the empty string.

                    1.   N3 > N4
                    2.   N4 < 1
                    3.   There are fewer than N3-1 distinct nonoverlapping
                         instances of S2 in S1.
                    4.   S2 is empty.

                    Provided that none of the above conditions holds,
                    the value of $PIECE is the contiguous substring of
                    S1 which is both to the right of the N3-1th occurrence
                    of S2 in S1 and to the left of the N4th occurrence
                    of S2 in S1.

                    The occurrences of S2 in S1 are counted from the left
                    end of S1 and are considered to be nonoverlapping.
                    Thus, there are two occurrences of "ABAB" in "ABABABABAB",
                    in the sense of the terms used here.

                    If N3 is 1 or less, the selected substring includes
                    the left end of S1. If N4 is greater than the number
                    of occurrences of S2 in S1, the selected substring
                    includes the right end of S1.

Cross-reference:

Examples:

Prior condition	Function value
SET X="ABC*DEF"	$P(X,"*",1)="ABC"
	$P(X,"*",2)="DEF"
	$P(X,"*",3)="" (the empty string)
	$P(X,"B",1)="A"
SET Y="B"	$P(X,Y,1)="A"
	$P(X,Y,2)="C*DEF"
	$P(X,"/",1)="ABC*DEF"
SET Z="A.B.C.D"	$P(Z,".",1)="A"
	$P(Z,".",2,3)="B.C"
	$P(Z,".",1,100)="A.B.C.D"
	$P(Z,".",3,2)="" (the empty string)
	$P(Z,"",1,100)=""

10.2.10

Name: $RANDOM

Abbreviation: $R

Syntax: $R[ANDOM] (integer expression)

Value: Let N denote the value of the argument. N must be
                         positive. $R(N) returns a random or pseudo-random
                         integer uniformly distributed in the closed interval
                         [0,N-1].

Cross-reference:

Examples:

1.   SET X=$RANDOM(N)   X will be given an integer value
                                        between 0 and N-1.

2.   SET Y=$R(2)   Y will randomly be given either
                                        the value 0 or 1.

3.   SET Y=$R(1)   Y is 0.

$SELECT

Name: $SELECT

Abbreviation: $S

Syntax:

$S[ELECT](truth-value expression:expression

                    [,truth-value expression:expression]...)

Value: Each argument of $SELECT is an ordered pair separated
                by a colon: truth-value expression:expression .
                The function may contain any positive number of arguments.
                At least one truth-value expression must have a value
                1 (true).

                Without evaluating the corresponding expression, each
                truth-value expression is evaluated, one at a time,
                in left-to-right order. This evaluation stops at
                the first argument for which the truth-value expression
                has the value 1. In this argument, and in no other,
                the expression is evaluated and this latter value
                becomes the value of $SELECT. Notice that no other
                expression need have a defined value.

Cross-reference:

Examples:

Prior condition	Function value
SET X=1	$S(X=1:8,2=3:0)=8
	$S(X=1:8,2=2:0)=8
	$S(X=2:8,2=2:0)=0
	$S(X=2:8,2=3:0)=error; one argument must have a true truth value.
SET Y="B"	$S(X=3:8,Y="B":"HELLO",X=1:13)="HELLO"
	$S(X=Y:B(1),Y="A":^(2,3),1:3)=3 Note that this function call does not use or alter the Naked Indicator.
SET ^A(1)="TEST"	$S(X=1:^A(1),Y:S)="TEST" The Naked Indicator is changed to ^A .

Name:	$TEXT
Abbreviation:	$T

Syntax:

$$\text{\$T[EXT]}\left(\begin{array}{l} \text{+ }\underline{\text{integer expression}} \\[6pt] \underline{\text{line reference}} \end{array} \right)$$

Value:

The argument denotes a line of the routine body which is at the top of the Partition Stack. This denotation is made in either of the following ways.

1. If the argument has the form + integer expression, its value, which we shall denote by N, must be greater than zero. The line denoted is the Nth line of the routine body. (The first line is associated with N=1.)
2. If the argument has the form line reference, the line is that denoted by the line reference, according to the method given in Chapter 7.

If the argument of $TEXT, by virtue of a line reference containing a label not appearing in a line head of the routine body or by virtue of a too-large value of an integer expression, does not denote a line, the value of $TEXT is the empty string. Otherwise, the value of $TEXT is the value of the line denoted, except for the following changes: ls is replaced by one space (SP), and eol is deleted.

Cross-reference:

Chapter 6   Partition Stack
Chapter 7   routine body
Chapter 7   line reference

Examples:

Assume that the following routine body is at the top of the Partition Stack.

```
ABC          ;SAMPLE
             SET X=3 WRITE Y
             READ X
Z            WRITE !,X
```

Then:

```
$T(ABC)="ABC  ;SAMPLE"
$T(Z)="Z WRITE !,X"
$T(ABC+1)=" SET X=3 WRITE Y"
$T(ABC+2)=" READ X"
$T(+1)="ABC ;SAMPLE"
$T(+3)=" READ X"
$T(Z+1)=""
$T(@A+2)=" READ X"  (given that A="ABC")
$T(ABC+K)=" READ X"  (given that K=2)
$T(+B)=" READ X"  (given that B=3)
```

The following two lines write out the routine body containing them.

```
F I=1:1 S X=$T(+I) Q:X=""  W X,!
W #
```

Name: $VIEW

Abbreviation: $V

Syntax: $V[IEW](argument syntax specified by implementor)

Value: $VIEW is an implementation-specific function available
                     to the implementor who may wish to make available
                     to programs certain data not otherwise available.
                     Each implementation must recognize and accept the
                     $VIEW function, regardless of any interpretation given
                     to it.

                     [Port: Routines designed for portability should not
                     contain calls of the $VIEW function.   :Port]

$Z

Name: $Zremainder of name specified by implementor

Abbreviation: $Zremainder of abbreviation specified by implementor

Syntax:

$Zspecified by implementor(argument syntax specified by implementor)

Value: Implementors desiring to offer functions not in the
                         standard are required by the standard to spell their
                         function names beginning with $Z.

                         [Port: Routines designed for portability should not
                         contain calls of any $Z function. :Port]

CHAPTER 11

EXPRESSIONS

Table of Contents

CHAPTER 11

EXPRESSIONS

11.1 General Rules Governing Expressions

11.1.1 Syntax of expression

An expression is a substring of a command which, when executed, yields
a value. The execution of expressions is the principal means by which
values are obtained for subscripts, for arguments of functions, and for
assignment to storage nodes.

Expressions may be arbitrarily complex, subject to certain practical limits
such as that placed on line length for the sake of portability (see Chapter 7);
however, there are typically just two atomic elements of which expressions
are built: expression atoms and binary operators. An expression atom
(discussed in Chapter 12) is the simplest form of an expression; it is
the atomic value-yielding substring of a line of which more complex expressions
are compounded. Binary operators provide the linkages between expression
atoms to form the more complex expressions.

As is almost every other process in the execution of a MUMPS routine,
the execution of an expression is a progressive process which accompanies
a left-to-right scan of characters of the line. This basic progressive
structure is revealed in the syntax definition of expression.

$$\text{expression} \quad ::= \quad \left| \begin{array}{l} \underline{\text{expression atom}} \\[2ex] \underline{\text{expression}} \ \underline{\text{expression tail}} \end{array} \right.$$

The right-hand side of the above recursive formulation is syntactically
equivalent to the following simpler, iterative, formulation

expression atom [expression tail] ...

but the recursive formulation is chosen to correlate more closely with
the calculation model which is presented. Every expression tail is in
two parts: a binary operator on the left and an expression atom or pattern
on the right.

11.1.2   MUMPS Values

The universe V within which all MUMPS values fall is the set of all strings
of characters chosen from the 7-bit ASCII set.   (See Appendix A for a
list of these 128 ASCII characters.) This universe includes the empty
string. Except for certain practical limits such as storage capacity,
given a value chosen from this universe, a MUMPS routine can be written
to produce that value.

[Port: The practical universe in which all values produced by MUMPS routines
designed for portability must fall contains all ASCII strings whose lengths
are 255 or less and no strings whose lengths exceed 255. :Port]

There are four important subsets of V: the subset Ve of numeric values
expressed in exponential notation, its subset Vn of numeric values without
exponents, its subset Vi of integer values, and its subset Vt of truth
values. These subsets form a descending nested sequence in the order
given. The subsets are defined by the syntax of the string which are
their elements. This relationship between set membership and syntax is
expressed in the following table.

Strings in the value subset	All have the syntax of	Defined in
Ve	exponential value	11.1.3
Vn	numeric value	11.1.2
Vi	integer value	11.1.2
Vt	truth value	11.1.2

The syntax definitions of numeric value, integer value, and truth value
are given below.

$$\text{truth value} \quad ::= \quad \begin{vmatrix} 0 \\ 1 \end{vmatrix}$$

$$\text{nonzero digit} \quad ::= \quad \begin{vmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{vmatrix}$$

$$\text{digit} \quad ::= \quad \begin{vmatrix} 0 \\ \text{nonzero digit} \end{vmatrix}$$

$$\text{integer value} \quad ::= \quad \begin{vmatrix} 0 \\ [-] \text{ nonzero digit } [\text{digit}] \ldots \end{vmatrix}$$

[Port: The number of digits of any integer value produced by the execution of a MUMPS routine designed for portability may not exceed nine. :Port]

$$\text{fraction value} \quad ::= \quad . \; [\text{digit}] \ldots \text{nonzero digit}$$

(Note that the single period denotes ".", whereas the group of three periods denotes optional repetition of the [digit] syntactic element.)

$$\text{numeric value} \quad ::= \quad \begin{vmatrix} 0 \\ [-] \text{ nonzero digit } [\text{digit}] \ldots [\text{fraction value}] \\ [-] \text{ fraction value} \end{vmatrix}$$

Each integer value or numeric value uniquely denotes a real number; the number denoted is the obvious one. The truth value 0 denotes False; the truth value 1 denotes True.

[Port: A MUMPS routine designed for portability may produce only those numeric values denoting zero and the numbers whose absolute values are in the closed interval

$$[10^{-25}, 10^{25}] \quad . \quad \text{:Port]}$$

11.1.3   Interpretation Operations

The value sets V, Ve, Vn, Vi, and Vt form a nested sequence of sets, as
depicted in the following Venn diagram.
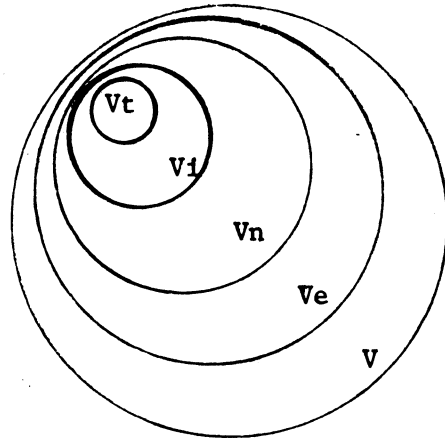
That is, every truth value is an integer value, every integer value is
a numeric value, every numeric value is an exponential value, and every
exponential value is a MUMPS value. Many operators, some examples of
which are listed here, have natural definitions, however, only on one
of these subsets of V but not on the whole of V.

1.   Arithmetic is defined on numeric values.
2.   The Boolean operators are defined on truth values.
3.   Certain functions, such as $CHAR and EXTRACT, have some arguments
     which are assumed to be integer values.

It is a principle of design of the MUMPS language that any operation which
is defined on a subset of V will not fail to operate if an actual argument
falls outside that subset. Rather, any operation which requires such
a restricted argument automatically performs an "interpretation" operation
on each such argument before the operation itself is performed. The
appropriate interpretation operation maps all MUMPS values into the required
subset (while not changing those values already having the correct syntax),
thereby guaranteeing that all argument values will meet the restrictions
assumed in the definition of the operation.

In each argument of a function, operator, command, etc., the requirement
for a particular interpretation operation is denoted by the syntactic
form of the argument.

Syntactic form of argument	Required subset of values	Interpretation operation performed on argument prior to execution
expression, expression atom	V	none
exponential expression, exponential expression atom	Ve	exponential interpretation Ie
numeric expression, numeric expression atom	Vn	numeric interpretation In
integer expression, integer expression atom	Vi	integer interpretation Ii
truth-value expression, truth-value expression atom	Vt	truth-value interpretation It

Note that in the routine itself, these specialized argument forms have
the same syntax as their more general forms.

exponential expression atom	::=	expression atom
numeric expression atom	::=	expression atom
integer expression atom	::=	expression atom
truth-value expression atom	::=	expression atom
exponential expression	::=	expression
numeric expression	::=	expression
integer expression	::=	expression
truth-value expression	::=	expression

Each interpretation operation is a mapping from the whole of V onto the required subset of values, which does not alter those values already in the required subset. This is expressed schematically as follows.

$$V \xrightarrow{\text{Ie}} Ve$$

$$V \xrightarrow{\text{In}} Vn$$

$$V \xrightarrow{\text{Ii}} Vi$$

$$V \xrightarrow{\text{It}} Vt$$
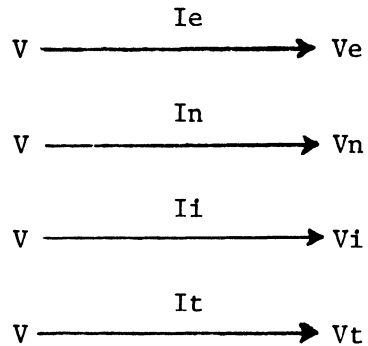
Whereas MUMPS arithmetic operators do not produce results in exponential or "scientific" form, the numeric interpretation In correctly interprets argument data in this form and converts it to numeric form. This is the reason for the introduction of the set Ve, even though exponential values per se are not employed in the definition of the arithmetic operations. (In Chapter 12, the exponential form of data is used more explicitly in connection with the interpretation of the numeric literal form of expression atom.) The syntax definition of exponential value, which defines the elements of Ve, follows.

digit sequence ::= digit [digit] ...

$$\text{mantissa} ::= \begin{vmatrix} \text{digit sequence } [\text{ . digit sequence }] \\ \text{ . digit sequence} \end{vmatrix}$$

exponent ::= E [$\overset{+}{\_}$] digit sequence

exponential value ::= [-] mantissa [exponent]

The interpretations In, Ii, and It then break down into sequences of more than one of the primitive interpretations Ie, Ien, Ini, and Int, as expressed in the following diagram and table.

$$
V \xrightarrow{\text{Ie}} Ve \xrightarrow{\text{Ien}} Vn \xrightarrow{\text{Ini}} Vi
$$

with

$$
Vn \xrightarrow{\text{Int}} Vt
$$
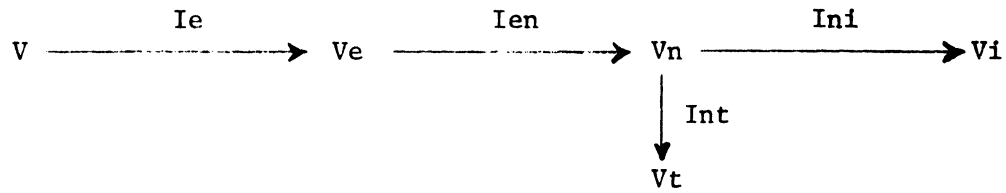
To obtain the interpretation	Apply the sequence
In	Ie followed by Ien
Ii	In followed by Ini
It	In followed by Int

11.1.3.1   The Interpretation Ie

The transformation of any data string S into one in Ve proceeds in two
steps.   The first simplifies any leading signs; the second takes the largest
head satisfying exponential value.

> <u>Definition of Head.</u>   The head of a string with respect to a
> given point which is immediately to the right or left of any
> character of the string is defined to be that substring which
> contains all characters to the left of the point and no characters
> to the right of the point.   Note that a head may be empty, or
> it may be the entire string.

1.   Apply the following initial-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

2.   Apply whichever of the two following cases is appropriate.

> a.   If the leftmost character of S is not "-", the result is
>   the longest head of S having the syntax of <u>exponential value</u>.
>
> b.   If S is of the form - T, take the longest head of S which
>   has the syntax of <u>exponential value</u>.   If the <u>mantissa</u> part
>   of the result denotes 0 (or -0), make the whole value "0".

11.1.3.2   The Interpretation Ien

1.   If the <u>mantissa</u> part of the argument has no ".", place one at the
      right end of the <u>mantissa</u>.

2.   The presence of an <u>exponent</u> is recognized by the "E". If it is absent,
      skip step 3.

3.   If the <u>exponent</u> has a + or has no sign, move the "." a number of
      decimal digit positions to the right in the <u>mantissa</u> equal to the
      number denoted by the <u>exponent</u>, appending zeros to the right of the
      <u>mantissa</u> as necessary. If the <u>exponent</u> has a minus sign, move the
      "." a number of decimal digit positions to the left in the <u>mantissa</u>
      equal to the absolute value of the number denoted by the <u>exponent</u>,
      appending zeros to the left of the <u>mantissa</u> as necessary.

4.   Delete the <u>exponent</u> and any leading or trailing zeros of the <u>mantissa</u>.

5.   If the rightmost character is ".", remove it.

6.   If the result is empty or "-0", make it "0".

11.1.3.3   The Interpretation Ini

1.   The presence of a <u>fraction value</u> is recognized by the ".". If it
      is present, delete the <u>fraction value</u>.

2.   If the result is empty or "-", make it "0".

11.1.3.4   The Interpretation Int

If the number is not "0", make it "1".

11.1.4   Calculation of <u>expression</u> Values

The syntax of an instance of <u>expression</u> has one of the two forms

<u>expression</u> <u>atom</u>

or

<u>expression</u> <u>expression tail</u> .

Chapter 12 prescribes the value of each <u>expression atom</u>. If the expression
has the form <u>expression atom</u>, then the value of the expression is the
value of the expression atom.

Otherwise, the expression is of the form

<u>expression</u> <u>expression tail</u>   ,

and the following general comments apply to its evaluation.

The MUMPS scan of an expression is left to right. At the time that the
expression is to be evaluated, the rightmost character of the expression
tail has just been scanned. Therefore, the values of all component expressions
and/or expression atoms are already known. Specifically, the value of
the left-hand <u>expression</u> is known, and the value of any <u>expression atom</u>
in the <u>expression tail</u> is known.

Because an evaluation occurs immediately after each expression tail is
scanned, and because the direction of scan is left-to-right, an expression
containing several expression tails, e.g.,
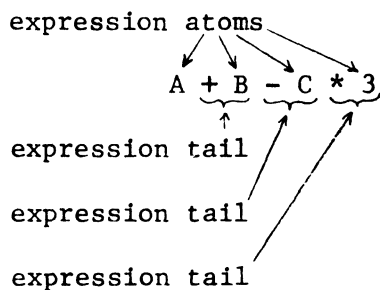
is naturally grouped from the left. That is, any implied parentheses
are as follows.

$$(((A + B) - C) * 3)$$

If it is desired to force a different order of calculation, parentheses
may be used to insure that certain combinations of symbols be treated
as expression atoms, to wit:
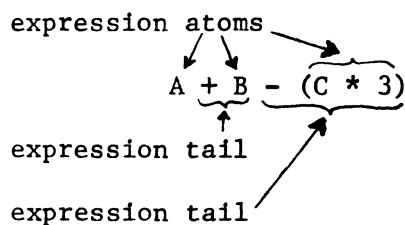
11.2   Types of Expression Tails

It may be predicted from examining just the binary operator of the following expression tail

<u>expression</u> <u>expression tail</u>

in what value class, Vt, Vi, Vn, or V, the value of the overall expression will fall. This fact is the basis for grouping the expression tails into the following four groups, each of which "restricts" its resulting value into the named value class.

1.   Nonrestricting expression tails. The only operator in this group is string concatenation (_).
2.   Numeric-value restricting expression tails. This group contains the arithmetic operators + - / * #.
3.   Integer-value restricting expression tails. The only operator in this group is the integer-quotient divide (\ ).
4.   Truth-value restricting expression tails. This group contains the relational operators (= < >] [ ), the logical operators (& !), and the pattern-match operator (?). Any of these operators may be immediately prefixed with the not-prefix (') which reverses the truth value of the result.

The syntax definition of <u>expression tail</u> follows.

$$
\underline{\text{expression tail}} \ ::= \ \left|\ \left|\ \begin{array}{l} \underline{\text{string operator}} \\ \underline{\text{arithmetic operator}} \\ ['] \ \underline{\text{truth operator}} \end{array}\right| \ \underline{\text{expression atom}} \right.
$$

$$
['] \ ? \ \underline{\text{pattern}}
$$

$$
\underline{\text{string operator}} \ ::= \ \_
$$

$$
\underline{\text{arithmetic operator}} \ ::= \ \left|\ \begin{array}{l} + \\ - \\ * \\ / \\ \backslash \\ \# \end{array}\ \right|
$$

$$
\underline{\text{truth operator}} \ ::= \ \left|\ \begin{array}{l} \underline{\text{relation}} \\ \underline{\text{logical}} \end{array}\ \right|
$$

$$
\underline{\text{relation}} \ ::= \ \left|\ \begin{array}{l} < \\ > \\ = \\ [\\ ] \end{array}\ \right|
$$

$$
\underline{\text{logical}} \ ::= \ \left|\ \begin{array}{l} \& \\ ! \end{array}\ \right|
$$

II.2.1 Nonrestricting Expression Tails

Operators: _ (string concatenation)

Expression syntax: **expression 1** _ **expression atom 2**

Value calculation: The value of the resulting expression is the result
of concatenating the value of __expression 1__ with the
value of __expression atom 2__, with __expression 1__ on the
left.

Cross-reference:

Examples: Let X have the value "DE".

Expression	Value
"A" "B"	AB
"A"_"B" "C"	ABC
"A"_"1"_23	A123
"AB"_X_(3*4)	ABDE12
1_2	12
1_2*3	36
1_(2*3)	16

11.2.2   Numeric-value Restricting Expression Tails

Operators: + (algebraic addition)
                          - (algebraic subtraction)
                          * (algebraic multiplication)
                          / (algebraic division)
                          # (modulo)

Expression syntax:

numeric expression 1 | + - * / # | numeric expression atom 2

Value calculation: [Port: Routines designed for portability should not
                     contain calculations requiring more than nine significant
                     digits of accuracy in any intermediate or final values.
                     :Port]

                     The numeric interpretation In is taken of both arguments.

                     The value of the result is defined by the real number
                     which it denotes, as follows.

                   + produces the algebraic sum.

                   - produces the algebraic difference.

                   * produces the algebraic product. [Port: The value
                     of the result may not be uniquely defined if the number
                     of significant digits in the product exceeds nine.
                     :Port]

/ 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 not permitted. [Port: The value of the
result may not be uniquely defined if the number of
significant digits of the quotient exceeds nine or
if the quotient contains a nonterminating fraction.
:Port]

produces the value of the left argument modulo the
right argument. # 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 $\leq$ x.
[Port: The value of the result may not be uniquely
defined if the number of significant digits in the
result exceeds nine. :Port]

Cross-reference: 11.1.3 Numeric interpretation In

Examples: Let A=0,B=1,C=2.

Expression	Value	
2+3	5	
2*3+4	10	
4+2*3	18	
(4+2)*3	18	
4+(2*3)	10	
4+(2/3)	4.66666667	(approximately)
1+.123456789	1.12345679	(approximately)
A-B-C	-3	
A-(B-C)	1	
B#C	1	
C#B	0	
B+2*C	6	
B+(2*C)	5	

11.2.3   Integer-value Restricting Expression Tails

<u>Operator:</u>   \ (integer-quotient divide)

<u>Expression syntax:</u>   <u>numeric expression 1</u> \ <u>numeric expression atom 2</u>

<u>Value calculation:</u>   The value of the result is the integer interpretation
                   Ii of the value of

<u>numeric expression 1</u> / <u>numeric expression atom 2</u> .

[Port: The value of the result may not be uniquely
defined if the number of significant digits in the
result exceeds nine. :Port]

<u>Cross-reference:</u>   11.2.2 division operator /

<u>Examples:</u>

<u>Expression</u>   <u>Value</u>

4+2\3   2
4+(2\3)   4

11.2.4   Truth-value Restricting Expression Tails

Operators: Relational operators (see 11.2.4.1)
            =   (string identity)
            <   (algebraic less than)
            >   (algebraic greater than)
            [(string contains)
            ]   (string follows)
        Logical operators (see 11.2.4.2)
            &   (and)
            !   (or)
        Pattern-match operator (see 11.2.4.3)
            ?

11.2.4.1   Relational Operators

Expression syntax:

$$\underline{\text{numeric expression 1}} \quad [\,'] \quad \begin{vmatrix} < \\ > \end{vmatrix} \quad \underline{\text{numeric expression atom 2}}$$

$$\underline{\text{expression 1}} \quad [\,'] \quad \begin{vmatrix} = \\ [\\ ] \end{vmatrix} \quad \underline{\text{expression atom 2}}$$

Value calculation: In the case of all of these operators, denoted here by op,

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

A op B has the value 1 if the relation it expresses is true; it has the value 0 otherwise.

The inequalities > and < compare the numeric interpretations of their arguments; the relation which they express is the conventional algebraic "greater than" or "less than".

The operator = yields the value 1 if and only if the two arguments (without taking any interpretation) are identical strings. If both arguments are known to be in numeric-value form, the uniqueness of the numeric representation permits = to be used to test for numeric equality. [Port: The effects of inexact arithmetic must be taken into account if any truncation has taken place in the calculation of an argument. :Port]

The relation [is called "contains". A [B is true if and only if B is a contiguous substring of A; that is, A [B has the same value as ''$F(A,B). 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 lexicographically in the conventional ASCII collating sequence. A is defined to follow 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 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 and the remainder of A follows the remainder of B.

11.2.4.2 Logical Operators

Expression syntax:

$$\underline{\text{truth-value expression 1}} \; [\,'\,] \; \left| \begin{array}{c} \& \\ ! \end{array} \right| \; \underline{\text{truth-value expression atom 2}}$$

Value calculation: & and ! are given the names "and" and "or", respectively.

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

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

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

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

11.2.4.3   Pattern-match operator

Expression syntax:

expression 1 ['] ? pattern

$$\text{pattern} ::= \left| \begin{array}{l} \underline{\text{pattern atom}}\ [\underline{\text{pattern atom}}]\ \dots \\[4pt] @\ \underline{\text{expression atom}}\ \underline{V}\ \underline{\text{pattern}} \end{array} \right|$$

$$\underline{\text{pattern atom}} ::= \left| \begin{array}{c} \underline{\text{integer literal}} \\ \bullet \end{array} \right| \left| \begin{array}{c} \underline{\text{string literal}} \\ \underline{\text{pattern code}} \end{array} \right|$$

$$\underline{\text{pattern code}} ::= \left| \begin{array}{c} C \\ N \\ P \\ A \\ L \\ U \\ E \end{array} \right| \dots$$

Value calculation: S '? P is defined to have the same value as '(S ? P).

A pattern is a concatenated list of one or more pattern atoms. Let n be the number of pattern atoms in pattern. S ? pattern is true if and only if there exists a partition of S into n substrings

$$S = S\underline{1}\ S\underline{2}\ \dots\ S\underline{n}$$

Such that there is a one-to-one order-preserving correspondence between the $S\underline{i}$ and the pattern atoms, and each $S\underline{i}$ "satisfies" its respective pattern atom. Note that some of the $S\underline{i}$ may be empty.

Each pattern atom consists of two parts: a pattern code or a string literal, which is used as the basis for a comparison with $S\underline{i}$, and a preceding multiplier: either a numeric literal specifying an exact number of iterations or the "indefinite multiplier" ".".

If the _integer literal_ multiplier is present, Si satisfies
the pattern atom if and only if Si is a concatenation
of _integer literal_ substrings, each of which satisfies
the string literal or pattern code of the pattern
atom.

If the indefinite multiplied "." is present, Si satisfies
the pattern atom if it is a concatenation of any number
(including zero) of substrings, each of which satisfies
the string literal or pattern code of the pattern
atom.

A substring satisfies a string literal if and only
if it is identical to the value of the string literal.

A pattern code is a string of letters, each of which
represents a class of characters, as follows.

Letter	Class of characters
C	33 Control characters, including DEL
N	10 Numeric characters
P	33 Punctuation characters, including SP
A	52 Alphabetic characters
L	26 Lower-case alphabetic characters
U	26 Upper-case alphabetic characters
E	Everything (the Entire set of characters)

The class of characters represented by a pattern code
is the union of the classes of characters represented
by each of the letters of the pattern code. A substring
satisfies a pattern code if and only if each of the
substring's characters falls into the class of characters
represented by the pattern code.

11.2.4.4   Examples of Truth-valued Operators

Relational and logical operators

Let X=3,Y="A"

Expression	Value
1=1	1
1=0	0
(X=3)!(1=1)	1
(X=3)+1	2
X=3+1	2
X>Y	1
X>Y=1	1
Y'<X=1	0
X=3!(Y="A")	1
Y="A"_"B"	1B
"MGH"]Y<3	1

Pattern-match operator

Prior condition	Expression	Value
SET X="123456"	X?.N	1
	X?2N4E	1
	X?1"12"4N	1
	X?.A	0
	X?.N1"6"	1
	X?.E1"3".N	1
SET X="A12B12C"	X?.N	0
	X?.NA	1
	X?2NA.E1"1".E	1
SET X="TESTTESTtest"	X?2"TEST".E	1
	X?1"TEST".E	1
	X?8U4L	1
	X?8A1"test"	1
	X?.U1"test"	1
	X?.A2L	1
SET X="12.34 dollars"	X?.N1P.E	1
	X?2N1P2N1" ".L	1
SET X=12.34	X?.N	0
SET X=1234.	X?.N	1
SET X="1234."	X?.N	0
SET X=$C(7)_$C(10)	X?2C	1
	X?.E1C.E	1

CHAPTER 12

EXPRESSION ATOMS

Table of Contents

CHAPTER 12

EXPRESSION ATOMS

12.1   Syntax of <u>expression atom</u>

An expression atom is the atomic value-yielding component of an expression.
(See 11.1.1 for a discussion of the role of expression atoms in expressions.)
When executed, an expression atom yields a value. This chapter describes
the syntax and value of each form of <u>expression atom</u>.

The syntax definition of expression atom follows. To the right, there
is a cross-reference to the detailed discussion of each form of expression
atom.

<table>
<tr><td></td><td></td><td></td><td>Defined in</td></tr>
<tr><td></td><td></td><td>(<u>expression</u>)</td><td>12.1.1</td></tr>
<tr><td></td><td></td><td><u>special variable</u></td><td>Chapter 9</td></tr>
<tr><td></td><td></td><td><u>function</u></td><td>Chapter 10</td></tr>
<tr><td></td><td></td><td><u>string literal</u></td><td>12.2</td></tr>
<tr><td><u>expression atom</u></td><td>::=</td><td><u>numeric literal</u></td><td>12.3</td></tr>
<tr><td></td><td></td><td><u>storage reference</u></td><td>12.4</td></tr>
<tr><td></td><td></td><td>' <u>truth-value expression atom</u></td><td>12.5</td></tr>
<tr><td></td><td></td><td>$\left|{+\atop-}\right|$ <u>numeric expression atom</u></td><td>12.5</td></tr>
</table>

12.1.1   Use of Parentheses to Form Expression Atoms

Any calculation procedure which may be expressed as an <u>expression</u> may
be used to compute the value of an expression atom by writing the expression
atom as (<u>expression</u>) . Since the value of an expression's component expression
atoms must be computed prior to the computation of the value of the expression,
this use of parentheses is the basis for controlling the order of evaluation
of the components of an expression. (Section 11.1.4 contains more discussion
and an example on this subject.)

12.2   String Literals

A "literal" is an expression atom whose value is a function only of its
spelling. There are two types of literals, string literals (discussed
here) and numeric literals (discussed in 12.3).

The presence of a string literal is indicated by the quotes which are
its first and last characters. To define the syntax of string literal
one needs to define the class of characters nonquote. In the interest
of space, this definition is an informal one. A nonquote is any single
ASCII graphic except for the quote sign (").

$$\underline{\text{string literal}} \quad ::= \quad " \quad \left[\begin{array}{c} " \ " \\ \hline \underline{\text{nonquote}} \end{array} \right] \ \ldots \quad "$$

In words, a string literal is bounded by quotes (considered to be part
of the literal), and it contains within the quotes any string of graphics,
except that when quotes occur inside, they occur in adjacent pairs. The
value of an instance of string literal is the same as its spelling except
that the following deletions are made.

1.   Neither of the two bounding quotes is in the value.
2.   Whenever a quote pair occurs inside the bounding quotes, one quote
     of the pair is deleted.

The string literal denoting the empty string is written   ""   .

12.3   Numeric Literals

The presence of a numeric literal (in a context in which an expression atom is appropriate) is indicated by its first character, which must be either a digit or a decimal point. A numeric literal is a form of literal, designed primarily for convenience in arithmetic calculation, whose value is restricted to Vn.

The syntax of numeric literal is that of exponential value, except that no leading sign is permitted.   (See 11.1.3.)

> numeric literal   ::=   mantissa [exponent]

The value of a numeric literal is derived from its spelling by direct application of the interpretation operation Ien (see 11.1.3.2).

The construct integer literal is used in other definitions within this manual.

> integer literal   ::=   digit [digit]   ...

The value of an integer literal is derived from its spelling by direct application of the interpretation operation Ii (see 11.1.3).

Examples of literals

Spelling	Value
"HELLO"	HELLO
""""HELLO""""	"HELLO"
1	1
0001	1
1.23	1.23
1E1	10
1E2	100
1.5E2	150
3.1415E3	3141.5
2.34E-2	.0234
.01E2	1
"0001"	0001

12.4   Storage References

A storage reference is a unique designator of a node of either Named Partition
Storage or Named System Storage. Storage references may appear in contexts
other than the expression atom context (for example, in the argument of
the $DATA function or to the left of the = in the SET command). Whenever
a storage reference appears in the expression atom context, its value
is the content of the Value attribute of the designated storage node;
in this context, if there is no storage node as designated by the storage
reference, or if the storage node exists but its Value attribute is undefined,
execution of the storage reference is considered to be in error, and it
has no defined value.

Syntactically, there are three varieties of storage reference, as enumerated
in the following syntax definitions.

$$\underline{\text{name}} \quad ::= \quad \left| \begin{array}{c} \% \\ \underline{\text{alpha}} \end{array} \right| \quad \left[\begin{array}{c} \underline{\text{digit}} \\ \underline{\text{alpha}} \end{array} \right] \quad \cdots$$

[Port: Routines designed for portability should not have any lower-case
letters in names. names with 9 or more characters should be distinguished
on the basis of their first 8 characters.   :Port]

$$\underline{\text{storage reference}} \quad ::= \quad \left| \begin{array}{l} \underline{\text{local variable}} \\ \underline{\text{global variable}} \\ \underline{\text{naked reference}} \end{array} \right|$$

$$\underline{\text{local variable}} \quad ::= \quad \left| \begin{array}{l} \underline{\text{name}} \; [(\underline{\text{expression}} \; [,\underline{\text{expression}}] \; \cdots \;)] \\[2ex] @ \; \underline{\text{expression atom}} \; \underline{V} \; \underline{\text{local variable}} \end{array} \right|$$

$$\underline{\text{global variable}} \quad ::= \quad \left| \begin{array}{l} {}^{\wedge}\underline{\text{name}} \; [(\underline{\text{expression}} \; [,\underline{\text{expression}}] \; \cdots \;)] \\[2ex] @ \; \underline{\text{expression atom}} \; \underline{V} \; \underline{\text{global variable}} \end{array} \right|$$

$$\underline{\text{naked reference}} \quad ::= \quad \left| \begin{array}{l} {}^{\wedge}(\underline{\text{expression}} \; [,\underline{\text{expression}}] \; \cdots \;) \\[2ex] @ \; \underline{\text{expression atom}} \; \underline{V} \; \underline{\text{naked reference}} \end{array} \right|$$

The expressions in parentheses are called "subscripts".

The syntactic form	Designates a node in
local variable	Named Partition Storage
global variable	Named System Storage
naked reference	Named System Storage

The LOCK command requires a name-only reference which excludes **naked reference**.
variable name exists for this purpose.

$$\underline{\text{variable name}} \quad ::= \quad \left| \begin{array}{l} \underline{\text{local variable}} \\[2ex] \underline{\text{global variable}} \end{array} \right|$$

The algorithm by which an instance of <u>storage reference</u> designates a storage node consists of the following two steps.

1. A mapping, called here the Node Designation Mapping (described below), is executed, using the storage reference instance as its argument. The value of this mapping is a <u>node designator</u> (see 4.2).
2. If a storage node exists whose Name attribute contains the same string as the computed node designator, this storage node is unique and it is the designated storage node. If it does not exist, the resultant action depends on the context in which the <u>storage reference</u> appears.

<u>Description of Node Designation Mapping</u>

The Node Designation Mapping starts with an empty Node Designator Register and builds up the final node designator by a process of successive concatenation to the right of the content of the Node Designator Register.

1. Call the Node Designator Register "NDR". The first step depends on the syntactic form of the <u>storage reference</u> or <u>variable name</u> instance.

    a. If <u>local variable</u>, place the instance of <u>name</u> into the NDR.
    b. If <u>global variable</u>, place the instance of [^]<u>name</u> into the NDR. (That is, the first character is "^".)
    c. If <u>naked reference</u>, place the content of the Naked Indicator into the NDR. If the Naked Indicator is undefined, the <u>storage reference</u> instance is erroneous.

2. If the <u>storage reference</u> or <u>variable name</u> instance is a <u>local variable</u> or <u>global variable</u> which contains no subscript, the Node Designation Mapping ends here. Otherwise, proceed to step 3.

3. Execute this step once for each subscript expression, proceeding in a left-to-right direction. This step contains the following two parts.

    a. First, evaluate the <u>expression</u>. Let V denote its value.
    b. Then concatenate ¢V to the right of the NDR.

4. This completes the Node Designation Mapping. If the argument is a <u>storage reference</u>, refer to 5.2 for a description of any effect on the Naked Indicator of executing a <u>global variable</u> or <u>naked reference</u>. If the argument is a <u>variable name</u>, there is no effect on the Naked Indicator.

[Port: MUMPS routines designed for portability may only execute subscripts whose values have the syntax of <u>nonnegative integer</u>.

$$\text{nonnegative integer} ::= \left| \begin{array}{l} 0 \\[4pt] \underline{\text{nonzero digit}} \; [\underline{\text{digit}}] \; \ldots \end{array} \right|$$

There may be at most nine digits in the value of a subscript expression. Accompanying each implementation should be a description of the interpreter's response to subscript expressions whose values are other than nonnegative integers with nine or fewer digits. Interpreters which accept such subscript values should do so in a way which maintains consistency with the rest of this reference manual. :Port]

<u>Examples of correct local variables</u>

```
X          AB          A(3)        AB(4,5,X)   A(B(3))
A(3,4)     AB(X,Y(Z),^A(3,4,5))    A(B(C(D(E,F(G)))))
A(1)       A(--1)      A(+1)       A("1")      A(1.0)
A(01)      A(1.5E1)    A(1.)       A(+"12A")
```

<u>Examples of nonportable storage references</u>

```
A(1.5)     A(-1)       A("01")     A(12A)      A("12A")
A(1.55E1)  A("1.")
```

<u>Examples of Naked References</u>

Prior content of Naked Indicator	Storage reference	Node designator	Subsequent content of Naked Indicator
undefined	^A(1)	^A¢1	^A
^A	^(2)	^A¢2	^A
^A	^(3,4)	^A¢3¢4	^A¢3
^A¢3	^(5)	^A¢3¢5	^A¢3
^A¢3	^B(3,5)	^B¢3¢5	^B¢3
^B¢3	^X	^X	undefined
undefined	^(2)	error	undefined
undefined	^Y	^Y	undefined
undefined	^X(5,6,7,8)	^X¢5¢6¢7¢8	^X¢5¢6¢7
^X¢5¢6¢7	^(9)	^X¢5¢6¢7¢9	^X¢5¢6¢7
^X¢5¢6¢7	^(9,-2)	^X¢5¢6¢7¢9¢-2 (*)	^X¢5¢6¢7¢9 (*)
^X¢5¢6¢7¢9	^("+")	^X¢5¢6¢7¢9¢+ (*)	^X¢5¢6¢7¢9 (*)

(*)Note: Some implementations may properly consider this to be erroneous; if they do not, this is the result they should show.

12.5   Unary Operators

The three operators '(not), +(plus), and -(minus) are called unary operators. Their effects are defined as follows.

The value of 'truth-value expression atom is the complement of the value of truth-value expression atom.   That is, where T is a truth-value expression atom,

$$'T \; = \; \begin{cases} 1 & \text{if the value of T is 0} \\ 0 & \text{if the value of T is 1} \end{cases}$$

The value of +numeric expression atom is the same as the value of numeric expression atom.   That is, + is simply a way of forcing execution of the numeric interpretation In.

The value of -numeric expression atom is the negative of the value of numeric expression atom.   Let V be the value of numeric expression atom. The value of -numeric expression atom is the result of applying In to the string -V.   (Simplification of double negatives occurs in the sign reduction rules of 11.1.3.1.)

Let unary operator denote any of the three unary operators.   The portion of the syntax description of expression atom referring to unary operators reads as follows.

    expression atom   ::=   unary operator expression atom

This recursive formulation is chosen over an iterative formulation expressed informally by

                    [unary operator] ... expression atom

in order to make explicit that the order of evaluation of unary operators is right-to-left.

Examples of unary operators

Expression	Value	
+"A"	0	
+-"A"	0	
-"5.0"	-5	
'"5.0"	0	
'(1+2E-10-1)	1	(possibly)
'(1-1+2E-10)	0	(definitely)
3--5	8	
3-0--5	8	
3---5	-2	
3--'"A"	4	

APPENDIX A

Table of ASCII Characters

The character notation is that used in ANS X3.4-1968. The code values
are those which appear as values of the $ASCII function and as arguments
of the $CHAR function.

Code	Character	Code	Character	Code	Character	Code	Character
0	NUL	32	SP	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	−	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124		
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

APPENDIX B

Index of Syntactic Types

Page numbers refer to principal definitions.

APPFNDIX C

Index of Technical Terms

MUMPS Development Committee Manuals

MDC Doc. No.	Identification
NBS Handbook 118	January 1976 (with errata sheets thru March 9, 1976), MUMPS Language Standard
	Part I: MDC/28, 3/12/75, MUMPS Language Specification M. E. Conway
	Part II: MDC/33, 9/17/75, MUMPS Transition Diagrams D. D. Sherertz and Anthony I. Wasserman
	Part III: MDC/34, 9/17/75, MUMPS Portability Requirements E. A. Gardner and C. B. Lazarus
29	5/28/75, MUMPS Interpreter Validation Program User Guide J. Rothmeier and P. L. Egerman
30	6/25/75, MUMPS Translation Methodology P. L. Egerman, C. B. Lazarus and P. T. Ragon
35	11/10/76, MUMPS Documentation Manual L. J. Peck and R. A. Greenes
1/11	6/13/75, MUMPS Primer M. E. Johnson and R. E. Dayhoff
2/1	5/15/75, MUMPS Globals and Their Implementation A. I. Wasserman, D. D. Sherertz and C. L. Rogerson
2/2	5/30/75, Design of a Multiprogramming System for the MUMPS Language A. I. Wasserman, D. D. Sherertz and R. W. Zears
2/3	6/15/75, Implementation of the MUMPS Language Standard A. I. Wasserman and D. D. Sherertz
3/5	8/31/76, MUMPS Programmers' Reference Manual M. E. Conway and P. L. Egerman