1995

STANDARD MUMPS POCKET GUIDE

FIFTH EDITION

FREDERICK D. S. MARSHALL

for Octo Barnett,
Bob Greenes,
Curt Marbles,
Neil Papalardo,
and Massachusetts General Hospital
who gave the world MUMPS

and

for Ted O'Neill,
Marty Johnson,
Henry Heffernan,
Bill Glenn,
and the MUMPS Development Committee
who gave the world standard MUMPS

THE 1995 STANDARD MUMPS POCKET GUIDE

FREDERICK D. S. MARSHALL

MUMPS BOOKS
• SEATTLE •
2010

THE 1995 STANDARD MUMPS POCKET GUIDE

FIFTH EDITION OF THE MUMPS POCKET GUIDE

SECOND PRINTING

MUMPS BOOKS

AN IMPRINT OF THE VISTA EXPERTISE NETWORK

819 North 49th Street, Suite 203 · Seattle, Washington 98103

www.vistaexpertise.net

info@vistaexpertise.net

(206) 632-0166

VISTA

EXPERTISE NETWORK

CONTENTS

1 · INTRODUCTION

1.1 · PURPOSE

This booklet summarizes the MUMPS programming language for MUMPS programmers seeking quick reference. It summarizes all the variables, operators, functions, commands, and other elements of MUMPS, except for those described in other standards (GKS, JIS90, MWAPI, OMI, SQL, TCP-IP, X3.64, and X-Windows).

1.2 · ACKNOWLEDGMENTS

This fifth edition, compiled by Frederick D. S. Marshall, reflects the 1995 MUMPS standard.

The following individuals reviewed and critiqued earlier drafts of this guide: Wally Fort, Kristi Hanson, Dave Holbrook, Sandra Reynolds, Beverly Marshall Saling, Kate Schell, George Timson, Maury Pepper, Larry Landis, Duglas Kilbride, and especially David Marcus. Thanks also go to Jack Bowie, Robert Greenfield, Dan Schullman, David Sherertz, Robert Stimac, George Timson, Tony Wasserman, and Jerry Wilcox, all of whom contributed substantially to previous editions.

Special thanks go to Thomas Salander, author of the fourth edition (1991); Joel Achtenberg, author of the third and second editions (1978 and 1983); and Joan Zimmerman, who wrote the original Pocket Guide under grant number HS-01540 from the National Center for Health Services Research, Department of Health, Education, and Welfare.

For an authoritative definition of MUMPS consult the standard itself: *X11.1-1995* by the Mumps Development Committee. The third edition of the 1995 standard, *ISO/IEC 11756:1999, Information Technology–Programming Languages–M,* is available from the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). The fourth edition, *MUMPS 1995,* will be published by Mumps Books in 2010.

The seventh and significantly improved edition of the MUMPS pocket guide, *MUMPS 1995 Pocket Guide,* will be published by Mumps Books in 2010.

For an in-depth description of MUMPS complete with extensive examples, consult the *MUMPS 1995 Reference Manual,* which will be published by Mumps Books 2011.

For a self-instruction workbook for students of MUMPS, consult the *MUMPS 1995 Workbook,* which will be published by Mumps Books in 2011.

For a textbook for teaching MUMPS, consult *M Programming: A Comprehensive Guide* by Richard Walters.

For information about these and other MUMPS references, contact Mumps Books, 819 North 49th Street, Suite 203, Seattle, Washington, USA, www.vistaexpertise.net, info@vistaexpertise.net, (206) 632-0166.

3 · THE SUITE OF STANDARDS

The MUMPS programming language is implemented according to a standard created and updated by the Mumps Development Committee (MDC), which then submits it for approval to the American National Standards Institute (ANSI), the National Institute of Standards and Technology (NIST), and the International Organization for Standards (ISO). The 1977 and 1984 standards were approved by ANSI, the 1990 standard was approved by all three bodies, and the 1995 standard standard was approved by ANSI and ISO.

Beginning with the 1995 standard, MUMPS was defined by a suite of standards:

X11.1: The M Programming Language

X11.2: The Open Mumps Interconnect

X11.3: The Graphical Kernel System

X11.4: The X-Window Binding

X11.6: The M Windowing Api (MWAPI)

The MUMPS standard describes some features by referring to other, independent standards, and explaining how to access those features in MUMPS. Such descriptions bind the MUMPS standard to other standards, rather than duplicate the work of those standards. The 1995 standard includes six bindings: GKS, JIS90, SQL (X3.135), TCP-IP, X3.64, and X-Windows.

The MUMPS standard describes most language elements in detail; for some it leaves certain portions undefined, left up to the implementer. Thus, although use of those features is standard, it is not portable across implementations. Developers wishing to write code that is completely portable across all MUMPS implementations should avoid the use of non-portable features.

Section 2 of the X11.1 standard defines the limits adhered to

by portable MUMPS code. Throughout this pocket guide, portability guidelines are given within square brackets [*like this*].

MUMPS implementers describe those parts of their implementations that do not meet the standard in a document called the conformance clause.

4 · SYSTEM MODEL

More than just a programming language, MUMPS provides a programming and execution environment, a database, a system of interfacing between multiple programming languages, and a multi-processing system. The MUMPS programming language is an imperative (command-oriented), dynamic, late-binding language oriented toward manipulation of strings and sparse arrays with string subscripts.

4.1 · MULTI-PROCESSING

MUMPS systems are designed around multi-processing, in which multiple users share the same computing resources. Each user of a MUMPS system is assigned a process with a unique ID ($JOB), within which the user can manipulate private or shared data. Each process includes a set of structured system variables and intrinsic special variables that provide the process with information about its characteristics and current status, a process stack to aid in executing code, storage for the currently executing routine, and a symbol table in which to store private data.

MUMPS provides ways to equitably share data, input/output devices, and other common resources, as well as built-in features that the programmer can use to synchronize the activities of separate processes (LOCK), to generate new processes (JOB), or to end the current process (HALT).

4.2 · DATA

Each process can create and manipulate its own private data in local variables. Unlike most programming languages, MUMPS extends the use of variables to include what other systems do with files; MUMPS systems store shared data in

global variables. Local variables are temporary, lasting only for the duration of the process, but global variables last indefinitely, beyond the life of any single process.

MUMPS variables need not be declared before use, but are instead created dynamically by giving them values. Any variable not yet created has an undefined value, and MUMPS provides two special functions for dealing with variables that may be undefined ($DATA and $GET). Attempts to retrieve directly an undefined value cause an error.

Variables are not statically typed, which would restrict them to storing only specific kinds of values; they are dynamically typed. Dynamic typing lets variables be of any data type at any time, changing types arbitrarily and without declaration during execution. MUMPS accommodates this by treating all data as variable-length strings, but interpreting those strings as the data types appropriate to the operations being performed upon them. Accordingly, MUMPS provides powerful built-in string manipulation features.

In addition, all variables can be subscripted arrays. These arrays are not statically declared, but are dynamically created and modified at run time.

Subscripts need not contain contiguous ranges of numeric subscripts, but instead can be sparsely populated with only the subscripts desired. Subscripts in MUMPS can have any string value, and are automatically collated.

Thus, sorting, which is usually a dominant activity in database software, is built into MUMPS data. MUMPS provides extensive array and tree manipulation features. As a result, MUMPS data structures are far more dynamic than in most languages, performing a significant portion of the computing usually reserved for code.

4.3 · CODE

Processes execute MUMPS code in modules called routines, each of which can execute other routines as needed. Additionally, MUMPS increases the flexibility of the code by blurring the distinction between code and data. Any line of code can be fetched as data ($TEXT), and assuming its value adheres to MUMPS code syntax, any variable may be executed as code (XECUTE). In addition, MUMPS routines can leave fragments of code unresolved until run time (@, indirection). This dynamic interaction between the code and data results in highly flexible code whose exact behavior is not bound until run time.

4.4 · ENVIRONMENTS

MUMPS systems collect code and data into environments. As each new MUMPS process is created it is assigned an environment, which dictates the capabilities and limitations of that process. Some routines and globals are accessible by other environments, while others are restricted. In this way environments supply both security and modularization to complex MUMPS systems.

4.5 · PACKAGES

MUMPS processes are not limited to MUMPS code in their own environments. Routines in other environments, or code external to MUMPS systems can be bound up into external packages callable by the MUMPS process, and code written in other languages can even be embedded within MUMPS routines.

4.6 · CHARACTER SETS

MUMPS can operate upon multiple character sets as well, reducing the language barrier between computer systems

and people of different languages. Each character set has its own collation system. The 1995 standard includes three character set profiles: M, ASCII, and JIS90.

Although each system has a default character set, the system's processes, routines, globals, and devices can all operate under different character sets, so that conceivably two people speaking different languages could work on the same MUMPS system in their own language.

4.7 · INPUT/OUTPUT DEVICES

Each MUMPS process is assigned a principal input/output device ($PRINCIPAL), but can gain ownership of others (OPEN), change at any time the current device for input and output (USE), identify the current device ($IO), and release ownership of devices (CLOSE). Due to the use of a single data type, the string, only two commands are needed for input (READ) and output (WRITE).

The READ and WRITE commands share some built-in device controls (such as form feed, line feed, and tabbing) for formatting output. However, additional device controls are control mnemonics bound up into mnemonic spaces. Devices can be opened under different mnemonic spaces that dictate the control mnemonics available at any time.

5 · SYNTAX

The MUMPS standard uses a metalanguage to describe its language features. Each element of this metalanguage is defined once in the standard, and then used by name thereafter. The following index describes each metalanguage element and identifies where in the XII.1 standard it is defined.

5.1 · METALANGUAGE ELEMENT INDEX

ELEMENT	DESCRIPTION, XII.1 PAGE #
: : =	metalanguage operator: definition, 6
[ ]	metalanguage operator: option, 6
| |	metalanguage operator: alternation, 6
. . .	metalanguage operator: repetition, 6
actual	actual argument, 47
actuallist	actual argument list, 47
actualname	actual argument name, 47
algoref	algorithm reference, 17
alternation	alternation, 42
argument	argument of a command, 44
binaryop	binary operator, 39
charset	character set, 67
charsetexpr	character set expression, 17
closeargument	CLOSE argument, 49
command	command, 43
commands	commands separated by cs, 8
commandword	command word, 43
comment	comment, 8
control	control character, 7
controlmnemonic	control mnemonic, 66
CR	carriage return character, 6
cs	command separator, 8

element	description, xii.1 page #
device	device, 18
deviceattribute	device attribute, 49
devicekeyword	device keyword, 49
deviceparam	device parameter, 49
deviceparameters	device parameters, 49
devicexpr	device expression, 18
digit	decimal digit character, 7
dlabel	indirect label (evaluated label), 46
doargument	do argument, 49
ecode	error code, 25
emptystring	empty string, 17
entryref	entry reference, 46
environment	set of distinct names, 15
eoffset	error offset, 38
eol	end-of-line, 7
eor	end-of-routine, 7
exfunc	extrinsic function, 23
exp	exponent, 21
expr	expression, 11
expratom	expression atom, 12
expritem	expression item, 21
exprtail	expression tail, 39
externalroutinename	external routine name, 47
externref	external reference, 47
extid	external identifier, 11
extsyntax	external syntax, 11
exttext	external text, 11
exvar	extrinsic variable, 23
fncodatom	$fnumber code atom, 31
fncode	$fnumbercode, 30
fncodexpr	$fnumber code expression, 30
fncodp	$fnumber code p, 31

element	description, xii.1 page #
fncodt	$fnumber code t, 31
FF	form feed character, 6
formalline	formal line (line with formal list), 8
formallist	formal argument list, 8
format	i/o format code, 65
forparameter	for argument, 51
function	intrinsic function, 29
glvn	global or local variable name, 12
gnamind	global name indirection, 15
gotoargument	goto argument, 52
graphic	graphic character (character with visible representation), 7
gvn	global variable name, 15
gvnexpr	global variable name expression, 18
hangargument	hang argument, 53
ident	identification character, 7
ifargument	if argument, 53
intexpr	expression, value interpreted as an integer, 23
intlit	integer literal, 21
jobargument	job argument, 53
jobparameters	job parameters, 53
killargument	kill argument, 54
L	metalanguage operator: list (list of), 6
label	label of a line, 8
labelref	label reference, 46
leftexpr	left expression, 61
leftrestricted	left restricted, 61
levelline	level line (line without formal list), 8
LF	line feed character, 6
li	level indicator, 8
line	line in routine, 7

element	description, xii.1 page #
linebody	line body, 8
lineref	line reference, 46
lname	local name, 54
lnamind	local name indirection, 12
lockargument	LOCK argument, 54
logicalop	logical operator, 41
ls	label separator, 8
lvn	local variable name, 12
mant	mantissa, 21
mergeargument	MERGE argument, 56
mnemonicspace	mnemonic space, 58
mnemonicspacename	mnemonic space name, 58
mnemonicspec	mnemonic space specifier, 58
name	name, 7
namevalue	name value, 28
newargument	NEW argument, 57
newsvn	NEW special variable name, 57
noncomma	non-comma, 25
nonquote	non-quote (any graphic character not equal to quote), 21
nref	name reference, 55
numexpr	expression, value interpreted numerically, 23
numlit	numeric literal, 21
openargument	OPEN argument, 57
openparameters	OPEN parameters, 58
packagename	package name, 47
patatom	pattern atom, 42
patcode	pattern code, 42
patnonY	pattern non-Y character, 42
patnonYZ	pattern non-Y-or-Z character, 42
patnonZ	pattern non-Z character, 42

ELEMENT	DESCRIPTION, XII.1 PAGE #
svn	special variable name, 24
system	system, 20
systemexpr	system expression, 20
textarg	$TEXT argument, 38
timeout	time-out specification, 45
transparameters	transaction parameters, 64
truthop	truth operator, 40
tsparam	TSTART parameter, 64
tstartargument	TSTART argument, 64
tvexpr	expression, value interpreted as a truth-value, 23
unaryop	unary operator, 27
useargument	USE argument, 65
V	metalanguage operator: evaluation, 6
writeargument	WRITE argument, 65
xargument	EXECUTE argument, 66

6.1 · ROUTINE STRUCTURE

MUMPS software is written in routines, each of which consists of a head and a body.

A routine head is the name of the routine followed by a carriage return and a line feed. The first character of the name must be either a letter (upper or lower case) or the "%" character; if the name is longer than one character, the remaining characters can be letters and digits (e.g., VanGogh, PICASSO3, %, %Hopper, or %42, but not 42, WYETH%, or 2BLAKE). [*For portability, routine names should be distinguished by their first eight characters and should not contain lower-case letters.*]

The routine body is one or more lines and a carriage return and line feed at the end of the routine. [*For portability, no routine may be more than 10,000 characters long.*]

6.2 · LINES

Each line in a MUMPS routine consists of a line-start indicator (one or more space), optionally preceded by a line label, followed by zero or more commands with their associated arguments, optionally followed by a comment, and the line must end with an end-of-line indicator (a carriage return and line feed).

Commands are separated by one or more spaces, but may have additional spaces between them. The minimum separation for an argumentless command is two characters. Spacing between the final command and the comment, when one occurs, is the same as between commands. If the line contains commands but no comment, no spaces can occur after the last command.

The first character of a comment is a semi-colon, and the

remaining characters are not treated as executable MUMPS code.

Line labels must be unique within a routine. A line label may be either an integer or have the same syntax as routine names have. Two or more lines in one routine starting with the same label cause error M57. [*For portability, labels, like routine names, should be distinguished by their first eight characters and should not contain lower-case letters.*]

Beyond these rules, all routine lines fall into one of two categories, level lines or formal lines, each of which has additional characteristics. Most lines are level lines. After the line-start indicator, level lines have zero or more level indicators, each of which consists of a period followed by zero or more spaces. The level of a level line is the number of level-indicator dots at the start of the line body plus one. Some examples of level lines are:

```
VANRIJN ; next line has an empty line body
Raphael
TITIAN  I PAINTER="Klee" D  Q
KLEE    . R DALI
GRECO   .   R KAHLO D
12      . . R KLIMT
%5      ..; what about Renoir?
```

Formal lines are used for parameter passing, and have a level of one (that is, no dots). They require the label, which is immediately followed by a list of formal parameters (an open parenthesis, a list of variable names, and a close parenthesis). The variables must be unsubscripted local variables. Having any variable named more than once in the same formal parameter list causes error M21. Formal lines must be called through an extrinsic or parameter passing; trying to do otherwise causes error M11. Some examples of formal lines are:

```
COUNT() Q ^ARTIST("COUNT") ; extrinsic variable
MONET(PAINTING) N YEAR D LOOKUP Q YEAR ; exfunc
%ADD(ARTIST) D NEW Q ; for use with DO or JOB
```

The standard suggests but does not require that the first line of a routine have a label equal to the routine name, have no commands, and have a comment that lists a contact, the routine's position within the hierarchy of its package, the date and time it was last changed, and/or a description. Here are some examples (see Annex E on page 101 of X11.1 for details):

```
KOLLWITZ ;
DEGAS    ;ART@VA.GOV,ART,FRENCH,IMPRESS
ESCHER   ;ART;19960210;query handler for MCE
```

[*For portability, the length of a routine line, including any label and comment, is limited to the MUMPS string-length limit of 255 characters. The carriage return and line feed at the end of each line are not counted in this limit.*]

6.3 · LINE REFERENCES

Whether identifying a starting place for execution or a location to examine, MUMPS code can refer to routine lines during execution. By controlling the kinds of line references allowed in certain operations, MUMPS restricts dynamic access into routines.

Label references can only identify code on lines that have labels, and must explicitly name either the label or the routine, possibly both. If only the routine is named, it implicitly refers to the first line of the routine, which must then have a label. If only a label is named, it implicitly refers to the current routine. If the process has access to another environment, it can include an environment in the routine reference (e.g., LIST^|"BAROQUE"|PAINTER). Referencing a non-existent environment causes error M26. References to a non-existent label cause error M13. DO and JOB commands that

pass parameters to the invoked code can only use label references, as can extrinsic function and variable names (e.g., $$COUNT^PAINTER but not $$DAVINCI+3^PAINTER)

Entry references expand on label references in two ways: by allowing label and routine names to be variable (through the use of indirection: @ARTIST^PAINTER, MCHLNGLO^@MEDIUM, or @ARTIST^@MEDIUM; see section 7.8.3 below), and by allowing the use of positive offsets to refer to lines without labels (e.g., @ARTIST+10^@MEDIUM, LIST+1^PAINTER, and PRINT+5). Use of a negative offset causes error M12. Offsets so large they do not refer to an existing line cause error M13. The DO and JOB commands when not passing parameters, and the GOTO command, can all use entry references, letting them execute code from any line of any routine.

Text references, used only by the $TEXT function, expand on entry references by removing the need to base line references on a label, letting $TEXT refer to lines in routines without knowing more than the name. Lines can be referred to as offsets from labels or as absolute line counts (e.g., +10, +1^@MEDIUM, or PAINT+9^MEDIUM, PAINT^MEDIUM), and these absolute references can be indirected in various ways (e.g., +LINE^@ROUTINE, and @TEXTREF). Absolute line counts less than zero cause error M5.

Code in external packages bound to MUMPS can be referred to by an external reference, which takes the form of an & character followed by an external routine name in the form either of a name or name^name (e.g., &PI). Optionally, a package name and a period may come before the external routine name (e.g., &MATH.PI). The DO command can use external references to invoke external code. The names of external variables and functions are also external references (see section 7.5).

[*External references are not portable.*]

6.4 · EXECUTION

MUMPS systems usually execute code in blocks of lines, but transaction processing and error processing change the rules of execution. The MUMPS Windowing API introduces a fourth execution mode, event processing, described in that standard. All four MUMPS execution modes depend upon the process stack.

6.4.1 · The Process Stack

The ability to stack context is essential to the execution of modular code. MUMPS implements it in three ways: the DO and XECUTE commands, references to extrinsic functions and variables, and error processing. All three, when they occur within a process, push another level on its process stack. The collection of information (called a frame) pushed on the stack is used to restore processing and state information when the new execution level finishes.

All three kinds of frames push onto the stack the execution level, and the execution location of the command or expression that created the new frame. The frames of extrinsics and argumentless DO commands also push the current value of the $TEST variable. So, for example, after completing an argumentless DO, $TEST regains the value it had before the DO command. Error frames push information about error conditions during error processing.

Variable scoping features of MUMPS, such as the NEW command, also use the process stack to save information for later recovery when the current execution level completes.

6.4.2 · Block Processing

Under normal conditions MUMPS processes code in blocks of lines at the same line level. Lines with a greater level are not part of the current block, and are ignored.

The execution frames for blocks are pushed on the stack in one of two ways. The execution frame at the first level of the process stack is pushed when the process is created, whether by signing onto a MUMPS system or through execution of a JOB command. Blocks of code can be nested by the DO and XECUTE commands or by references to extrinsic variables or functions; nesting pushes the execution frames of the new blocks onto the process stack.

[*For portability, MUMPS code should not use more than 127 levels of the process stack.*]

When a block's execution frame is pushed on the stack, the block is executed in a simple flow. The lines within each block are executed one at a time, beginning with the first line in the block and continuing down sequentially. Within each line, commands are executed left to right.

Four commands modify the flow of execution within a block. IF and ELSE conditionally skip execution of the rest of the line. FOR repeats execution of the rest of the line. GOTO repositions the flow at the start of another line within the block.

Execution frames are popped from the process stack when their code blocks end. Blocks automatically end after execution of the last line in the block, or they can be ended in mid-block by the QUIT command, when used outside the scope of a FOR command.

When an execution frame is popped from the stack, execution resumes where it left off, in mid-expression or mid-line, depending on how the block was called. In the case of extrinsics, the block was called during the evaluation of the expression that contains the extrinsic variable or function, so MUMPS uses the extrinsic's returned value to continue expression evaluation.

Note that GOTO commands at line level one can reposition

the flow of execution to the start of any other line at level one, even in other blocks or other routines. Taken to extremes, this can result in a flow of execution not particularly block-oriented, though the block processing rules of execution still apply.

6.4.3 · Error Codes

Code	Meaning
M1	naked indicator undefined
M2	invalid combination with $FNUMBER code atom P
M3	$RANDOM seed less than 1
M4	no true condition in $SELECT
M5	line reference less than zero
M6	undefined local variable
M7	undefined global variable
M8	undefined intrinsic special variable
M9	divide by zero
M10	invalid pattern match range
M11	no parameters passed
M12	invalid line reference (negative offset)
M13	invalid line reference (line not found)
M14	line level not 1
M15	undefined index variable
M16	argumented QUIT not allowed
M17	argumented QUIT required
M18	fixed length READ not greater than zero
M19	cannot copy a tree or subtree into itself
M20	line must have a formal parameter list
M21	algorithm specification invalid
M22	SET or KILL to ^$GLOBAL when data in global
M23	SET or KILL to ^$JOB for non-existent job number
M24	change to collation algorithm while subscripted local variables defined

Code	Meaning
M26	non-existent environment
M27	attempt to rollback a transaction that is not restartable
M28	mathematical function, parameter out of range
M29	SET or KILL on structured system variable not allowed by implementation
M30	reference to global variable with different collating sequence within a collating algorithm
M31	control mnemonic used for device without a mnemonic space selected
M32	control mnemonic used in user-defined mnemonic space which has no associated line
M33	SET or KILL to ^$ROUTINE when routine exists
M35	device does not support mnemonic space
M36	incompatible mnemonic spaces
M37	READ from device identified by the empty string
M38	invalid structured system variable subscript
M39	invalid $NAME argument
M40	call-by-reference in JOB actual parameter
M41	invalid LOCK argument within a transaction
M42	invalid QUIT within a transaction
M43	invalid range value ($X, $Y)
M44	invalid command outside of a transaction
M45	invalid GOTO reference
M57	more than one defining occurrence of label in routine
M58	too few formal parameters

The simplest expression in MUMPS is a variable, a string literal, a numeric constant, or a function (functions are discussed in sections 7.4, 7.5, and 7.7). Examples of each of these four types of expression are respectively:

```
LASTNAME
"Distaso"
7
$L(LASTNAME)
```

Such simple expressions are called atomic expressions. More complicated expressions can be built up by linking a number of atomic expressions by means of the arithmetic and other types of operators. For example:

```
SUM/TOTAL
SEX="Female"
"adult"_"hood"
```

MUMPS expressions are not evaluated in the order used by most programming languages. Most languages follow arithmetic operator precedence rules: first apply roots, logs, and exponentiation, second multiplication and division, and third addition and subtraction. Although nearly a universal standard, these rules do not declare the precedence of expressions that include relational or string manipulation operators as well as arithmetic.

To extend these rules of precedence to account for all the combinations that can arise in MUMPS expressions would yield too complex a system to remember. Accordingly, MUMPS has its own system of expression evaluation:

Rule 1) All MUMPS expressions are processed from left to right. Whereas 3+4*2 in most languages would equal 11, in MUMPS it equals 14.

Rule 2) MUMPS precedence is overridden through the use

of parentheses, which nest expressions. Thus 3+(4*2) equals 11 in MUMPS.

These precedence rules apply to all expression elements in MUMPS, not just operators. 3+(4*$E(268))*$T is either 11 or 0, depending on the value of $TEST; intrinsics, extrinsics, and externals have exactly the same precedence as any operator.

7.1 · VALUES

Strings are interpreted as other data types by rules that permit a unique representation of those data types as strings, and rules that dictate how to interpret those strings as the data types they represent.

7.1.1 · Representation

MUMPS represents constant values as one of two kinds of literals, either string or numeric.

String literals can represent any data type. A string literal is bounded by quotes (e.g., "Mozart") and contains any string of printable characters. Embedded quotes are represented as two consecutive quotes (e.g., "Beethoven's ""Eroica"" Symphony"). Each quote pair represents a single quote in the value denoted by the string literal. An empty string is represented as exactly two quotes (i.e., "").

[*For portability, character strings should not be longer than 255 characters, and should only include characters from a standard character set.*]

Numeric literals are a special case of string literals, a shorthand for representing numeric, integer, or truth values. A numeric literal has a mantissa optionally followed by the letter E and a positive or negative integer exponent; the mantissa can be either an integer or a real number (e.g., 3.14159, -42, 6.0225E23, .6E+2, or 5E-1, but not 0.34, or 2.00).

[*For portability, numbers should fall within the exclusive*

interval [-10^{25} , -10^{-25}] or [10^{-25} , 10^{25}] or be zero, and should rely on no more than fifteen significant digits.]

7.1.2 · Interpretation

Because data has only one base type, the string, MUMPS programmers never need to convert their data between types. Instead, data is interpreted appropriately as required by context. If a string does not equal the standard representation of the type it is being interpreted as, it is automatically coerced into the correct representation. MUMPS programmers use this type coercion to transform their values as needed.

Numeric interpretation involves taking the leftmost portion of the string that is either exponential (42E0), decimal (-2.718), or integer (1865) in form. It produces canonic numbers (e.g., the numeric interpretation of "0.34" is .34). If the string does not begin with a valid numeric representation, the numeric interpretation is zero.

Integer interpretation is formed from the numeric interpretation by dropping any fraction.

Truth-value interpretation is false if the numeric interpretation is 0; otherwise it is true.

Here are some interpretations of different strings:

String	Numeric	Integer	Truth Value
"810"	810	810	1
"98 POUNDS"	98	98	1
""	0	0	0
" 35"	0	0	0
"86+9"	86	86	1
"PAGE 10"	0	0	0
"-8.4"	-8.4	-8	1
"86E-1"	8.6	8	1
"---9"	-9	-9	1
"-0"	0	0	0

7.2 · VARIABLES

There are three kinds of variables: local, global, and special.

7.2.1 · Local Variables

Locals are stored in each process's symbol table, which lists the names, subscripts, and values of all locals currently defined for that process. Attempting to evaluate an undefined variable causes error M6. Their names have the same syntax [*and portability restrictions*] as routine names. Access to locals is restricted within a process through the use of parameter passing or the NEW command; locals not explicitly scoped by one of these methods are available to all routines executing within the process for the life of the local.

[*For portability, variable names, like labels, should be distinguished by their first eight characters and should not contain lower case letters.*]

[*For portability, the total space used by the local variables for a process must not exceed 10,000 characters. The total length of a local variable reference with all its subscripts must not exceed the string length limit. The values of subscripts are restricted only by the possible values of strings.*]

7.2.2 · Global Variables

Global names have the same syntax [*and portability restrictions*] as locals; global names are always preceded by a leading caret (^) symbol, as in ^MTA. Attempting to evaluate an undefined global variable causes error M7. Access to globals is restricted by environment.

A naked global reference is a shorthand syntax for specifying a global variable by omitting the variable name and possibly some of the subscripts. The first subscript in the subscript list of a naked global reference implicitly refers to the last subscript level of the most recent global reference.

Thus, if a reference has been made to ^x(1), a subsequent naked reference to ^(2,3) would access the value of ^x(2,3).

Note that *most recent global reference* includes any reference to any global, with only highly specialized exceptions (see the $NAME function and the LOCK command). Properly determining the most recent global reference depends upon a sophisticated understanding of how MUMPS commands are executed, so naked references should be used with caution.

Under certain special conditions, the naked indicator is not defined, and a reference to it will cause error M1. These conditions arise after:

1. a process begins, but before the first full global reference executes;
2. an unsubscripted full global reference executes;
3. a transaction rolls back;
4. the default global environment changes; or
5. $QUERY operates upon a global.

7.2.3 · Special Variables

MUMPS provides four kinds of special variables designed to give additional information to the process: intrinsic special variables and structured system variables (both of which give each process identification and status information), extrinsic variables (which let MUMPS programmers add new special variables to reflect the needs of their software), and external variables (which provide status information related to external packages). Attempting to evaluate an undefined special variable causes error M8. With few exceptions, special variables cannot be modified.

7.3 · STRUCTURED SYSTEM VARIABLES

Identifiable by the ^$ characters that begin their names,

structured system variables are arrays that describe the current status of the system. The name of any structured system variable may be abbreviated as specified in the standard (e.g., ^$c).

Each describes the characteristics of some system entity or resource. The first subscript always identifies the entity or resource described by the nodes underneath it. Therefore, in the list of nodes defined within each variable, only the second and subsequent subscripts are shown.

The ^$JOB nodes that permit assignment of the default environment for globals, locks, and routines, are settable. Setting them to a non-existent environment causes no error, but the attempts to refer to globals, locks, or routines in that environment will cause error M26.

7.3.1 · List of Structured System Variables

^$CHARACTER (*character-set-profile name*)

Defines the available character sets. [*Only M, ASCII, and JIS90 are portable.*]

`^$C("M","INPUT","JIS90")` => *xform to M from JIS90*
`^$C("M","OUTPUT","JIS90")` => *xform from M to JIS90*
`^$C("M","IDENT")` => *character validation*
`^$C("M","PATCODE","U")` => *U pattern code definition*
`^$C("M","COLLATE")` => *collation algorithm*

^$DEVICE (*device ID*)

Defines the available I/O devices. [*Device IDs are not portable.*]

`$O(^$D(""))` => *first device ID*
`^$D("SCK$5030","CHARACTER")` => *device's character set*
`^$D("SCK$5030",`*attribute*`)` => *attribute's current value*

^$GLOBAL (*global variable name*)

Defines the global directory.

```
^$G("^DD","CHARACTER") => global's character set
^$G("^DD","COLLATE") => its collation algorithm
```

^$JOB (*process ID*)

Defines the current processes.

```
$O(^$J("")) => first process ID
^$J(42,"CHARACTER") => process's character set
^$J(42,"GLOBAL") => process's global environment
^$J(42,"LOCK") => process's lock environment
^$J(42,"ROUTINE") => process's routine environment
```

^$LOCK (*name reference*)

Defines the currently held locks.

```
$O(^$L("")) => first locked name reference
$D(^$L("^DD(1)")) => true if ^DD(1) is currently locked
```

^$ROUTINE (*routine name*)

Defines the routine directory.

```
$D(^$R("DIC")) => true if DIC routine exists
^$R("DIC","CHARACTER") => routine's character set
```

^$SYSTEM (*system ID*)

Defines the current MUMPS system.

```
^$S("LIVE","CHARACTER") => system's default char set
```

^$Z . . . (*vendor-specific value*)

Defines vendor-specific system entities. [*Structured system variables whose names begin with ^$Z are not portable.*]

```
^$ZSPECIFIC("nonstandard") => nonportable
```

For more information on structured system variables, see chapter 7.1.3 of *MUMPS 1995*.

7.4 · EXTRINSIC VARIABLES & FUNCTIONS

Identifiable by the $$ characters that begin its name, an extrinsic variable is a special variable added to the language by a MUMPS programmer. Its name must be a label reference to the code that returns the value of the extrinsic variable (with an argumented QUIT command); for example, $$NOW^TIME might always equal the current time in human-readable format.

Similarly, an extrinsic function is a function added to the language by a MUMPS programmer, and its name must have the same format as that of an extrinsic variable. It takes a list of arguments and returns a value based on those arguments. The arguments are passed as parameters (see 8.1.4, Parameter Passing, below) to the code that computes its value, and the computed value is returned with an argumented QUIT command; for example, $$WEEKDAY^TIME("February 14, 1996") would return "Wednesday", and $$WEEKDAY^TIME("February 16, 1996") would return "Friday".

7.5 · EXTERNAL VARIABLES & FUNCTIONS

Identifiable by the $& characters that begin its name, an external variable is a special variable added to the language by an external package. The rest of the name must be a valid external reference, such as $&MATH.PI.

Similarly, an external function is a function added by an external package. An external function name has the same syntax as an external variable name, and must be followed by the list of arguments, such as $&MATH.SIN(VALUE).

[*External variables and functions are not portable.*]

7.6 · INTRINSIC SPECIAL VARIABLES

Identifiable by the $ character that begins its name, an intrinsic special variable has a unique name that may be abbreviated to its initial letter or letters (e.g., $ECODE or $EC but not $E). Intrinsic variables modifiable by the NEW or SET commands are so indicated below.

7.6.1 · List of Intrinsic Special Variables

$DEVICE

$D = Status of current device. Settable. One to three pieces separated by commas; if it evaluates to true (1), the device is in an error condition.

$ECODE

$EC = List of current error codes surrounded by commas. Settable. MDC errors begin with M, implementor errors with Z, user errors with U. For example, "" means no errors, but ",M6," means a reference to an undefined local variable occurred, and error processing is now in effect.

$ESTACK

$ES = Counts process stack levels. Newable. Newing it also sets it to 0.

$ETRAP

$ET = MUMPS code that will execute if an error occurs (e.g., S $ET="D DEBUG^ERROR"). Settable & Newable. Newing $ETRAP stacks its current value but leaves it set to that value.

$HOROLOG

$H = Current date and time as "days,seconds". The first number counts the days elapsed since December 31, 1840 at

midnight, and the second number counts the seconds since the last midnight (e.g., "56665,31576" was February 22, 1996 at 8:46:16 a.m.).

$IO

$I = Current I/O device.

$JOB

$J = Current process ID. A unique positive integer.

$KEY

$K = The control-sequence that terminated the current device's last Read command. Settable.

$PRINCIPAL

$P = Principal I/O device.

$QUIT

$Q = True (equals 1) if the current context was invoked as an extrinsic variable or function; false (0) otherwise. When $QUIT is true, the QUIT from the current block of code must have an argument.

$STACK

$ST = Current level of the process stack.

$STORAGE

$S = Characters of free space available for use.

$SYSTEM

$SY = Current system ID.

$TEST

$T = Result of previous timeout or argumented IF. Boolean. 1 means previous IF or timed action succeeded.

$TLEVEL

$TL = Current number of nested transactions.

$TRESTART

$TR = Number of times the current transaction has been restarted.

$X

$X = Approximate horizontal cursor or carriage position of current device. Settable.

$Y

$Y = Approximate vertical cursor or carriage position of current device. Settable.

$Z…

All implementation-specific intrinsic special variable names begin with $Z. Settable and Newable only if allowed by implementor. [*Special variables whose names begin with $Z are not portable.*]

For more information on intrinsic special variables, see chapter 7.1.4.10 of *MUMPS 1995.*

7.7 · INTRINSIC FUNCTIONS

Identifiable by the $ character that begins its name, an intrinsic function has a unique name that may be abbreviated to its initial letter or letters (e.g., $ASCII or $A but not $AS). All intrinsic functions take a list surrounded by parentheses of

one or more arguments separated by commas. All the arguments are evaluated before the function is executed, even if (as in the case of $GET when applied to defined variables) some arguments are not needed for the function to correctly execute.

7.7.1 · List of Intrinsic Functions

$ASCII

Ascii number corresponding to one character in a string. The first argument is the string to examine. The optional second argument is the position within the string of the character whose ASCII code should be returned, and defaults to 1.

```
$A("Beethoven") => 66
$A("HAYDN",3) => 89
```

$CHAR

Characters corresponding to a list of ASCII values. Each argument gives the code for one character (a negative integer yields an empty string).

```
$C(66,-1,65,67,72) => "BACH"
```

$DATA

Number indicating whether a variable is defined or has nodes. The argument is the variable to evaluate.

```
>K BORN ; $D(BORN) => 0
>S BORN=1797 ; $D(BORN) => 1
>S BORN(1)=1840 ; $D(BORN) => 11
>K BORN S BORN(0)=1 ; $D(BORN) => 10
```

$EXTRACT

One or more characters from a string. Can be used as destination of SET command. The first argument specifies the

source string. The optional second argument specifies the position of the first character to return, and defaults to 1. The optional third argument specifies the position of the last character of the substring to return, and defaults to the value of the second argument.

```
$E("Brahms") => "B"
$E("Handel",5) => "e"
$E("Mozart",4,6) => "art"
```

$FIND

Position of character following leftmost occurrence of substring in a string. The first argument is the string to search. The second argument is the substring to find. The optional third argument is the character position within the string from which to begin the search, and defaults to 1.

```
$F("SIBELIUS","I") => 3
$F("SIBELIUS","I",3) => 7
$F("SIBELIUS","I",7) => 0
```

$FNUMBER

Number formatted according to codes. The first argument is the number to format. The second argument is a string of codes that describe the formatting to perform (see examples). Note that the only code that can be combined with "P" (or "p") is ","; including it with any of the others causes error M2.

```
$FN(-42,"P") => "(42)"
$FN(42,"P") => " 42 "
$FN(-42,"T") => "42-"
$FN(42000,",") => "42,000"
$FN(42,"+") => "+42"
$FN(-42,"-") => 42
$FN(42000,"p,",2) => " 42,000.00 "
```

$GET

Value of a variable, or a default value if variable is not defined. The first argument is the variable whose value should be returned. The optional second argument is the value to return if the variable is undefined, and defaults to "".

```
>K BORN ; $G(BORN) => ""
$G(BORN,"unknown") => "unknown"
>S BORN=1841 ; $G(BORN) => 1841
$G(BORN,"??") => 1841
$G(BORN,$$ABC) => 1841
```

$JUSTIFY

Right justified string in a field of spaces. The first argument is the string to justify. The second argument is the number of character positions to use to right justify the string; if the length of the string itself exceeds this number, $JUSTIFY has no effect. The optional third argument is the maximum number of fractional digits to return, and defaults to the number of digits in the first argument.

```
$J("3.14159",9) => "  3.14159"
$J("3.14159",9,2) => "     3.14"
```

$LENGTH

Length of a string, measured in characters or pieces. The first argument is the string to evaluate. The optional second argument is the delimiter to use to partition the string for counting; if absent, the length is counted in characters.

```
$L("Verdi & Wagner") => 14
$L("Verdi & Wagner","&") => 2
$L("Verdi & Wagner"," ") => 3
```

$NAME

Evaluated name of a variable with some, all, or no subscripts;

such a string is called a name value. The first argument is the name to evaluate. The optional second argument is the maximum number of subscripts to return, and defaults to the number of subscripts in the first argument. A negative second argument causes error M39.

```
>S YEAR=1860
$NA(BORN(YEAR,7)) => "BORN(1860,7)"
$NA(BORN(YEAR,7),1) => "BORN(1860)"
$NA(BORN(YEAR,7),0) => "BORN"
```

$ORDER

Next or previous subscript in a specified array. The first argument is the name of the array, with its last subscript being the one to traverse. The optional second argument determines whether the previous (-1) or next (1) subscript value should return, and defaults to 1.

```
>K BORN S (BORN(1810),BORN(1809))=""
$O(BORN("")) => 1809
$O(BORN(1809)) => 1810
$O(BORN(1810)) => ""
$O(BORN(1810),-1) => 1809
$O(BORN(1809),1) => 1810
```

$PIECE

Partitions a string into pieces based on a delimiter, and returns some of those pieces. Can be used as destination of SET command. The first argument is the string to partition. The second argument is the delimiter to use to partition it. The optional third argument is the first piece of the string to return, and defaults to 1. The optional fourth argument is the last piece of the string to return, and evaluates to the third argument.

```
>S B3="Beethoven,Bach,Brahms"
```

```
$P(B3,",") => "Beethoven"
$P(B3,",",3) => "Brahms"
$P(B3,",",1,2) => "Beethoven,Bach"
>S $P(B3,",",2)="BRUCKNER"
B3 => "Beethoven,BRUCKNER,Brahms"
```

$QLENGTH

Number of subscripts in a variable name, passed as a name value (see $NAME). The argument is the name value to evaluate.

```
$QL("BORN(1675,5)") => 2
```

$QSUBSCRIPT

Specified part (name, environment, or a subscript) of a variable name, passed as a name value. The first argument is the source name value. The second argument is the part of the name value to return: -1 returns its environment (if one is present), 0 returns its unsubscripted name, and positive integers return the corresponding subscripts (e.g., 2 returns the second subscript).

```
>S NAME="^|""MUSIC""|DOB(1862,1)"
$QS(NAME,-1) => "MUSIC"
$QS(NAME,0) => "^DOB"
$QS(NAME,1) => 1862
$QS(NAME,2) => 1
```

$QUERY

Next subscripted variable name in array, returned as a name value. The argument is the name value of the starting subscripted variable.

```
>K BORN S BORN(1891)=""
>S BORN(1882,3)=""
$Q(BORN) => "BORN(1882,3)"
```

```
$Q(@"BORN(1882,3)") => "BORN(1891)"
$Q(BORN(1891)) => ""
```

$RANDOM

Random integer uniformly distributed over an interval between 0 and maximum-1, inclusive. The argument is a number one greater than the maximum value to return. An argument value less than 1 causes error M3.

```
$R(9) => a number between 0 and 8
```

$REVERSE

Characters of a string in reverse order. The argument is the string to reverse.

```
$RE("Brahms") => "smharB"
$RE("level") => "level"
```

$SELECT

Value corresponding to first true condition of list, evaluated left to right. Each argument is a conditional expression, followed by a colon, and then the expression whose value $SELECT should return if the corresponding condition evaluates to true. Arguments after the first with a true condition are not evaluated. If all conditions are false, error M4 occurs.

```
>S SYMPHONY=5
$S('$D(SYMPHONY):"?",1:"!") => "!"
>K SYMPHONY
$S('$D(SYMPHONY):"?",1:"!") => "?"
```

$STACK

Information about how a level of the process stack was created, what code is executing at that level, and what errors have accumulated there. The first argument is the level of the

stack to analyze, and returns a value that indicates how the execution frame at that level was pushed onto the stack: if due to a command, the full upper-case name of the command; if due to an extrinsic, the value "$$"; and if due to an error, its error code.

If the first argument equals -1, $STACK returns the depth of the stack. If 0, an implementation-specific value that indicates how this process was started.

The optional second argument returns other information about a stack level. The value "ECODE" returns a list of an error codes added at that stack level. "MCODE" returns the current line of MUMPS code (or value, in the case of an XECUTE command) at that stack level. "PLACE" returns the location of the current command at that stack level.

This example shows the values returned by $STACK for a hypothetical error situation:

```
$ST(-1) => 3
$ST(0) => "JOB"
$ST(1) => "DO"
$ST(2) => "$$"
$ST(3) => ",M6,"
$ST(2,"ECODE") => ",M6,"
$ST(2,"MCODE") => " K OOPS W !,OOPS"
$ST(2,"PLACE") => "FUMBLE+9^FOO +2"
```

$TEXT

A line of code from a routine. The argument is a reference to the line to return. Use of an absolute line count, as in the third example, but with a count less than zero causes error M5.

```
>S LINE="FUMBLE+9^FOO"
$T(@LINE) => "K OOPS W !,OOPS"
$T(+0^FOO) => "FOO"
$T(^FOO) ; first line of FOO
```

$TRANSLATE

A translation of a string, in which certain characters are removed or replaced. The first argument is the string to translate. The second argument is a string of the characters to remove (or replace if a third argument is present). The optional third argument is a string of the characters with which to replace the characters of the second argument in the first argument. If the second argument is longer than the third, the excess characters at the end of the second argument are removed, not replaced.

```
>S MAN="SHOSTAKOVICH"
$TR(MAN,"H") => "SOSTAKOVIC"
$TR(MAN,"HS") => "OTAKOVIC"
$TR(MAN,"O","U") => "SHUSTAKUVICH"
$TR(MAN,"OS","CU") => "CHUCTAKUVICH"
$TR(MAN,"SHO","E") => "EETAKVIC"
$TR(MAN,"AKVSHOT","MUS") => "MUSIC"
>S UP="ABCDEFGHIJKLMNOPQRSTUVQXYZ"
>S LO="abcdefghijklmnopqrstuvwxyz"
$TR(MAN,HI,LO) => "shostakovich"
```

$VIEW

Implementation-specific information. $VIEW is a standard function, but its arguments and returned values are not defined by the standard.

$Z…

All implementation-specific intrinsic function names begin with $Z. [*Special variables whose names begin with $Z are not portable.*]

For more information on intrinsic functions, see chapter 7.1.5 of *MUMPS 1995*.

7.8 · OPERATORS

MUMPS operators provide arithmetic, string manipulation, and indirection. They are recognizable (though not uniquely, since syntactical delimiters in the language are similar) as one or more punctuation characters. Remember that MUMPS operator precedence follows expression-evaluation rules (see the start of section 7).

7.8.1 · LIST OF OPERATORS

Arithmetic Unary Operators

+

Numeric coercion.

```
+"27.3 days" => 27.3
```

-

Negate numeric coercion.

```
-"3.14 radians/second" => -3.14
```

Arithmetic Binary Operators

+

Sum

```
2.718+"2 above e" => 4.718
```

-

Difference

```
2.12-"6.3 eV" => -4.18
```

*

Product

```
1.00794*"2 atoms/H2" => 2.01588
```

/

Division. Division by zero (using any of the three division operators) causes error M9.

```
144.132/12.011 => 12
```

\

Integer division

```
82.8\"29.5 years/orbit" => 2
```

#

Modulo division. A#B = A - (B*floor(A/B)), where floor(x) = the largest integer not greater than x.

```
42#5 => 2
-42#5 => 3
42#-5 => -3
-42#-5 => -2
```

* *

Exponentiated value. Results producing complex numbers are not defined (e.g., -1**.5). [*For portability, exponentiation has only seven significant digits of accuracy.*]

```
4**2 => 16
4**.5 => 2
4**-1 => .25
4**-2 => .0625
4**-.5 => .5
```

Arithmetic Relational Operators

<

Less than

```
1642<1879 => 1
```

>

Greater than

```
1452>1564 => 0
```

String Binary Operators

_

Concatenates

```
"Feynman: "_1918 => "Feynman: 1918"
```

String Relational Operators

=

Equals

```
1969-5=1964 => 1
1967="1967: M" => 0
1966=01966 => 1
1966="01966" => 0
"Lovelace"="Hopper"+2 => 2
"Lovelace"=" Lovelace "+2 => 2
```

[

Contains (note: by definition all strings contain the empty string "")

```
"Darwin"["win" => 1
```

]

Follows

```
"COPERNICUS"]"KEPLER" => 0
```

]]

Sorts after (its behavior depends on the current character set's

collation algorithm; e.g., in M numbers collate numerically, whereas in ASCII they collate as strings).

```
1683]]170 => 1 in M collation
1683]]170 => o in ASCII collation
"PRIESTLEY"]]"LAVOISIER" => 1
```

?

Pattern matches (see pattern match section below for details)

```
"Leakey"?1A => 0
"Boaz"?1.A => 1
"Fossey"?1U1.5L => 1
"Goodall"?.4L.P6C.E => 0
"Piaget"?1"Pi"4.U => 0
"Skinner"?2A1(1"in",78C).2U => 0
"Harlow"?.(1"Har",.6"Mas")1.AP => 1
"Maslow"?.E1"low".CNP => 1
```

Logical Operators

&

And

```
"Watson"&"Crick" => 0
"Morgan"&1735 => 0
1838&1839 => 1
-12000&1996 => 1
1859&0 => 0
```

!

Or

```
"Jenner"!"Pasteur" => 0
"Hoffman"!1928 => 1
1898!-400 => 1
1867!0 => 1
```

'

Not

```
'"Turing" => 1
''"Babbage" => 0
'"Backus"&1957 => 1
'("Wirth"&"Codd") => 1
"Wirth"'&"Codd" => 1
'("ALGOL"!1959) => 0
"ALGOL"'!1959 => 0
"Chen"'="Jacquard" => 1
"Ada Lovelace"'["Love" => 0
"ENIAC"'<"Naur" => 1
```

Special Operators

@

Indirect (see indirection section below for details)

```
>S BORN(-300)="EUCLID"
>S NAME="BORN"
@(NAME_"(-300)") => "EUCLID"
@NAME@(-300) => "EUCLID"

>S TAG="FUMBLE",ROUTINE="FOO"
>D @TAG^@ROUTINE

>S ARG="BORN(1856)=""FREUD"""
>S @ARG

>S PATTERN="1U4L"
"Curie"?@PATTERN => 1
```

For more information on operators, see chapters 7.1.4.11 and 7.2.1 of *MUMPS 1995*.

7.8.2 · Pattern Matching

Perhaps the second most powerful operator in MUMPS (for the most powerful, see 7.8.3, Indirection, below), pattern match tests whether a string matches a certain pattern. Pattern matches in *MUMPS 1995* have most of the capabilities of regular expressions, a widely-used standard for characterizing strings, but also are capable of operating on strings from any character set.

Each pattern consists of a series of pattern atoms, each of which must match part of the string (matching an empty string within a string is valid). Each pattern atom has two parts: a repetition count that declares how many substrings are required, and a pattern element that describes the substrings.

For example, the pattern 1U4L has two pattern atoms: 1U and 4L. The 1U requires one upper case letter, and the 4L four lower case, which is why "Curie"?1U4L evaluate to 1.

The repetition counts can be counts or ranges. Ranges use the "." character to specify the range, and can be open at either, both, or neither end. For example, 1U, 1.2U, .2U, and 1.U all have valid repetition counts. An upper bound less than the lower causes error M10.

Pattern elements can be pattern codes. For the M and ASCII character sets, the available codes are A, C, E, L, N, P, and U, matching to alphabetic, control, every character, lower case, numeric, punctuation, and upper case, respectively. Pattern codes can include any number of these characters in any order, such as 1.AP and .CNP.

Pattern elements can also be literal substrings, as in "Maslow"?.E1"low".CNP.

Finally, pattern elements can be alternations that contain choices of how to satisfy the pattern atom, for example, the pattern match "Harlow"?.(1"Har",.6"Mas")1.AP has an

alternation in its first pattern atom. Whenever an alternation is repeated, as is allowed here by the "." repetition count, each choice is independent of the previous ones; "", "Har", "HarHar", and "MasHar" would all satisfy the first pattern atom. In the 1995 standard, the choices within an alternation are restricted to pattern atoms, and may not include full patterns (although the next MUMPS standard will permit full patterns within alternation).

For flexibility, patterns may be defined at run time through indirection, as explained more fully below.

7.8.3 · Indirection

Indirection, the most powerful operator in MUMPS, is more an underlying capability that spans multiple syntaxes than it is a true operator. This is the ability to leave fragments of code unresolved; an expression stands in for the code fragments, and is evaluated and then resolved into code and executed at run time. The advantage of indirection is generalization, the capacity to handle classes of problems rather than just specific ones. In three out of five cases, the operator or delimiter @ identifies the indirection.

In *value indirection,* common to all programming languages, variable names are manipulated rather than values, so that the software can manipulate any data that satisfies its constraints. The presence of a variable name within an expression identifies the value indirection, so no @ is needed (e.g., NAME and ^SCIENCE("GOULD")).

In *name indirection* the names themselves can vary. Since names are used throughout MUMPS for many different purposes, there are many different kinds of name indirection. All are identified by the presence of the @ character, and all are recursive in multiple ways.

In variable indirection, the name of a local or global variable can be specified indirectly:

```
>S A="Curie",B="A",C=@B S D="@B",E=@D
>S F="B",G=@@F S ^A="Bohr",H=@("^"_B)
>S ^B=@$E(A)
```

In these examples, A, C, E, G, and ^B are all set to "Curie", and ^A and H to "Bohr". In all cases, the expression following the @ is evaluated, and the resulting value treated as a new variable name; in the case of E, the resulting name itself involves indirection, and so begins the resolution of indirection anew.

Subscript indirection reduces the need to use concatenation when writing code that traverses the subscripts of an array specified by indirection:

```
>S DIAM("EARTH")="12750 km",ROOT="DIAM"
@(ROOT_"(""EARTH"")") => "12750 km"
@ROOT@("EARTH") => "12750 km"
>S ROOT2=$NA(@ROOT@("EARTH"))
>S @ROOT2@("MILES")="7920 mi"
```

The first use of indirection in this example is variable indirection, while the others are subscript indirection. Subscript indirection can add subscripts to any level of array reference; in the second and third examples we add an "EARTH" subscript to the name DIAM, whereas in the fourth we add "MILES" to DIAM("EARTH").

Variable-name indirection is a specialized form of variable indirection that permits only unsubscripted locals, and does not evaluate the final name to get a data value. It is used by the KILL, NEW, and TSTART commands, and when passing parameters by reference, e.g., D REMOVE^FILE(.@NOBEL) and N (@SYS).

Lock-name indirection is a similar but distinct specialization of variable indirection used only for the LOCK

command. It permits subscripted or unsubscripted locals or globals, and the global references may or may not include an environment (e.g., S ADD="^SCIENCE(""Hawking"")" L +@ADD).

Label indirection and routine indirection let a command refer to a label or routine at run time dynamically, so the command can refer to multiple labels or routines.

Pattern indirection lets the pattern used in a pattern match operation vary dynamically at run time, and also uses the @ delimiter to identify the indirection. For example:

```
>S DATA(1)="Democritus",PATTERN(1)="1.A"
>S DATA(2)="430 A.C."
>S PATTERN(2)="1.8N1"" "".1(1"A.D.",1"B.C.")
>S VALUE=1 ; DATA(VALUE)?@PATTERN(VALUE) => 1
>S VALUE=2 ; DATA(VALUE)?@PATTERN(VALUE) => 0
```

Argument indirection lets the argument of a specific command or function vary dynamically at run time, and also uses the @ character. Every command that accepts arguments, except the FOR command, will accept a list of one or more of its arguments indirectly. For example:

```
>S IND="COMPUTER=""Mark 1""" S @IND
>S IND="DATE=1944,C=DATE_"":""_COMPUTER" S @IND
```

Most functions don't need argument indirection because their arguments are of some type that already indirects, such as a variable name. Functions that take multiple arguments, unlike commands, cannot indirectly refer to more than one argument at once; arguments must be indirected individually. Even $TEXT, whose argument is not of a type that normally accepts complete indirection, can be referred to indirectly.

Only $SELECT violates this pattern; it does not let a complete argument, a *condition:value* pair, be indirected. Instead, the condition and value must be indirected

independently, and the colon delimiter cannot be indirected at all.

Command indirection lets the resolution of entire commands be deferred until run-time. It is identified not by an @ character, but by its invocation with the XECUTE command. See the description of XECUTE in chapter 8, Commands, for details.

```
>S ACTION="S BORN(1966)=1" X ACTION
```

8 · COMMANDS

Mumps commands turn the potential of its data structures and the evaluation of its expressions into directed activity.

8.1 · GENERAL RULES

8.1.1 · Basic Syntax

Each command name has an abbreviation that may be used instead of the full name. Some commands take arguments; others don't. Those that do are separated from their argument by a single space, and can usually accept multiple arguments, separated by commas; execution of multiple arguments acts like separate instances of the command with a single argument each. Command names are case-insensitive (e.g., HALT, halt, and hALt are the same command).

8.1.2 · Post-conditionals (Command/Argument)

Many commands (see definitions in section 5.5) accept post-conditionals (expressions appended to the command name by a colon). Unless a post-conditional evaluates to true, the command does not execute or evaluate its arguments (therefore, the naked reference is unaffected by the command's argument, for example). Post-conditionals do *not* change $test.

Some branching commands accept post-conditionals on their arguments. Unless these conditions evaluate to true, the command skips those arguments.

8.1.3 · Timeouts

Four commands accept timeouts, which look like argument post-conditionals but evaluate instead to the maximum number of seconds to wait for the command to succeed.

Timeouts always set $TEST to indicate whether the timed command succeeded within the allotted time. [*For portability, the effects of timeouts lasting for non-integer durations, e.g. 2.5, should not be relied upon.*]

8.1.4 · Parameter Passing

Extrinsic function references and the DO and JOB commands can pass values or variables to the blocks of code they invoke. The list of values to pass must follow the name of the function or the argument of the DO or JOB command, and must be enclosed in parentheses (e.g., $$ADD^MATH(LENGTH,PAUSE) or J PRINT^SCIENCE("EINSTEIN")). Each value in this list can be an arbitrary expression. The list itself is called the actual parameter list.

At the receiving end, the referenced label within the routine must contain a list of formal parameters at least as long as the actual list. Trying to pass more actual parameters than there are formal parameters causes error M58. These formal parameters must be unsubscripted local variable names.

During invocation, when the execution frame of the new code block is pushed on the process stack, the actual parameters are bound to the formal parameters either by value or by reference (note that the JOB command can only pass parameters by value).

When parameters are passed by value, the corresponding named formal parameters are first *newed* (see NEW command) and then *set* equal to the values passed in the actual list. If the actual parameter is also a variable, it is independent of the formal parameter; changes to the formal variable do not affect the actual parameter. When the block completes and its execution frame pops off the stack, the formal parameters are

also popped, and any pre-existing local variables of the same names are restored to their previous states.

When parameters are passed by reference, the corresponding named formal parameters are added as new names for the same variables. That is, the name-table entries for the formal parameters point to the same data-table entries that the actual names do. After the block completes and its frame pops off the stack, the formal names for the variables are unbound from the actual variables and returned to their former bindings (if any), but changes to the values of the formal parameters within the completed block are reflected in the values of the actual parameters.

Whether parameters are passed by value or reference is decided by the caller, not the called block of code. The names of variables passed by reference are preceded in the actual list by a period (e.g., D UPDATE^SCHEDULE(.PATIENT)); a period is *not* placed before the name of the corresponding formal parameter name within the routine that defines it.

Parameters can be passed to external routines as well, by value or reference.

8.2 · LIST OF COMMANDS

BREAK

BREAK:*postcondition*

Stops execution of current process for debugging until signaled. [*Behavior and arguments not specified by the standard.*]

```
>B
```

CLOSE

CLOSE:*postcondition* Device:*parameters*, . . .

parameters =>

device parameter

(device parameter:*device parameter: . . .*)

device parameter =>

vendor-specific expression

device keyword

device attribute=expression

Releases ownership of an I/O device. CLOSE can also use device parameters to manipulate the device as it is released. If the current device is closed, the special variable $IO will be empty or reset to a default value.

[*Implementation-specific CLOSE parameters are not portable.*]

```
>C "AUDOUT","VIDOUT"
>C:OFFLINE "SCREEN1":("READY":6)
```

DO

DO:*postcondition*

DO:*postcondition* DoArgument:*postcondition, . . .*

Do Argument =

entry reference

label reference(actual parameters)

external reference(actual parameters)

Executes a subroutine, then returns control to the next command after the DO; for multiple arguments, each subroutine in turn is executed. The line referenced in each argument must have a line level of one, or the DO will cause error M14. Argumentless DO executes the following block of code, with a line level one greater than the DO's line level, then returns.

```
>D
>D:AUTHOR="SHAKESPEARE"
>D HAMLET,MACBETH,OTHELLO,KINGLEAR
>D ^PLAY:CHARACTR="FALSTAFF",LIST^PLAY
>S WORK="TEMPEST" D @WORK^PLAY
>S WORK="MUCHADO^PLAY" D @WORK
```

```
>D ADD^PLAY("LOVE'S LABOUR'S LOST")
```

ELSE

ELSE

Lets rest of the line execute only if $TEST evaluates to false.

```
>E
```

FOR

FOR

FOR LocalVariable=ForParameters

For Parameters =>

For Parameter,*For Parameter, . . .*

For Parameter =>

expression

numeric expression:numeric expr.:*numeric expr.*

Repeats execution of the rest of the line, and sets the value of a variable each time. Argumentless FOR repeats execution of the rest of the line without setting a variable. A QUIT or GOTO command terminates a FOR loop on the current line; QUIT terminates only the most recent in a series of nested FOR loops on the line, whereas GOTO terminates all active FOR loops on the line. Argument indirection not allowed.

Each FOR parameter defines a series of one or more values for the variable to accept, and executes the rest of the line once for each value in that series. Each FOR parameter can be either a single evaluated expression, or a range of numeric values. Ranges include a starting value, an increment with which to calculate subsequent values, and an optional maximum value. A FOR loop that runs out of values stops without needing a QUIT or GOTO.

```
>F
>F TALE=1:1:53 => executes 53 times
>F TALE=5:2:53 => executes 25 times
```

```
>F TALE=1:1 Q:TALE=53
>F TALE=1:1 G TELL:^NAME(TALE)="PARDONER"
>F TALE="WIFE OF BATH",12:1:13
>F TALE=2:2:8,11:2:21,24,27,28,31
>F AUTHOR="CHAUCER","BOCCACCIO"
>F TALE=1:1:53 F LINE=1:1 Q:'^TEXT(TALE,LINE)
```

GOTO

GOTO*:postcondition* EntryReference*:postcondition, . . .*

Transfers execution to a different line of code, without returning when that block of code completes. Trying to GOTO a line at a different line level, or trying to cross block boundaries when the GOTO's line level is greater than one, causes error M45.

```
>G:AUTHOR="DANTE" ^CANON
>G INFERNO:BAD,PARADISO:'BAD
>G VIRGIL^GUIDE:'BOOK3,BEATRICE^GUIDE
>S CANTO=26,BOOK="INFERNO" G @CANTO^@BOOK
```

HALT

HALT*:postcondition*

Ends the process. Releases all locked names, closes all opened devices, and aborts all active transactions. HALT never takes an argument.

```
>H
```

HANG

HANG*:postcondition* NumericExpression*, . . .*

Suspends execution of the process for approximately the specified number of seconds. Negative numbers or zero do not stop execution. HANG always takes an argument. [*For portability, the effects of hanging for non-integer durations, e.g. 2.5, should not be relied upon.*]

```
>H 3
>H TIME
```

IF

IF

IF TruthValueExpression, . . .

Lets the rest of the line execute only if all arguments evaluate to true; sets $TEST to whether the IF succeeded. Argumentless IF lets the rest of the line execute only if $TEST = 1. Note that because IF with multiple arguments is identical to multiple, independent IF commands, later arguments are evaluated only if earlier ones succeed.

```
>I
>I AUTHOR="CERVANTES"
>I FRIEND1="QUIXOTE",FRIEND2="SANCHO"
>I NAME="ALDONZA"!(NAME="DULCINEA")
>I $D(VIRTUE),VIRTUE="FRIENDS" D VIRTUE^CANON
```

JOB

JOB:*postcondition* JobArgument:*JobParams:timeout, . . .*

Job Argument =>

entry reference

label reference(actual parameters)

Job Parameters =>

vendor-specific expression

(expression:*expression: . . .*)

Makes an independent process that begins execution at the specified line of code. Timed JOB sets $TEST to whether the JOB command succeeded in the time specified. Note that the JOB command can only pass parameters by value; trying to pass by reference causes error M40. [*Implementation-specific JOB parameters are not portable.*]

```
>J ^LISTPLAY
```

```
>J PRINT^CANON:25 E  Q
>J PRINT^ESSAY("MONTAIGNE","OF EXPERIENCE")
```

KILL

KILL*:postcondition*

KILL*:postcondition* KillArgument*, . . .*

Kill Argument =>

local, global, or structured system variable name

(local variable name*,local variable name, . . .*)

Removes specified variables, and all their array descendants. Argumentless KILL removes all local variables, and their array descendants. An exclusive KILL, a KILL argument in the form of a parenthesized list of local variables, removes all local variables and their descendants, except those listed within the parentheses and their descendants.

```
>K
>K PLAY,AUTHOR,^CANON("MOLIERE","TEMP")
>K (AUTHOR,WORK)
```

LOCK

LOCK*:postcondition*

LOCK*:postcondition Sign*LockArgument*:timeout, . . .*

Sign => + or -

LockArgument =>

Name Reference

(NameReference*,NameReference, . . .*)

Gets and/or releases ownership of names. Argumentless LOCK releases ownership of all names held by current process. Names, like devices, can only be owned by one process at a time. Ownership of a name includes all array descendants of that name.

Locked names are logical names that look like local and global names for convenience only. A name used as an

argument of a LOCK command does not change the naked reference, nor can it be referred to by the naked reference.

Unlike devices, names can be used when not owned; locking is a voluntary signaling mechanism, not a method for preventing access. Locking a name does not prevent other processes from accessing or manipulating variables with the same names, only from successfully locking those names. MUMPS code should voluntarily lock names, especially global names, whose simultaneous use by multiple processes would lead to problems, such as loss of database integrity.

Each non-incremental LOCK argument (those without + or -) first releases all names owned by this process, and then gets ownership of the name specified. Incremental and decremental LOCK arguments get (+) or release (-) ownership of the specified names without releasing any other names. A name incrementally locked multiple times must be decrementally locked an equal number of times to release it.

A LOCK of a list of names enclosed by parentheses will not succeed until all names listed are simultaneously available. A nonincremental LOCK of a list of names will not release the names owned by the process until it can simultaneously get ownership of all the names in its list.

Timed LOCK sets $TEST to whether it got ownership of the name in the time requested; network latency and other extraneous delays are not counted in the timeout period. For arguments without a timeout, LOCK waits indefinitely until the name is released by whatever process owns it. Improperly coded, this can result in a deadly embrace, in which two processes stop execution at LOCK commands because each owns names for which the other is waiting.

```
>L
>L ^CANON("MILTON")
>L (CRITIC,^CANON("JOHNSON"))
```

```
>L ^CANON("GOETHE"):2
>L +^CANON("AUSTEN")
>L -^CANON("WORDSWORTH")
```

MERGE

MERGE*:postcondition* Variable=Variable, . . .

Variable =>

global, local, or structured system variable

Copies the value and all array descendants from one variable to another variable. If either variable is an array descendant of the other, it causes error M19.

```
>M WHITMAN=^CANON("WHITMAN")
```

NEW

NEW*:postcondition*

NEW*:postcondition* NewArgument, . . .

New Argument =>

local variable name

(local variable name,*local variable name, . . .*)

$ESTACK

$ETRAP

Saves and temporarily removes locals and their array descendants, and restores them when this block of code ends. Argumentless NEW saves and temporarily removes all locals and their array descendants, and restores them when this block of code ends.

Only unsubscripted local variables or the intrinsic special variables $ESTACK and $ETRAP may be used in the argument of the NEW command. An exclusive NEW, a NEW argument consisting of a list of names within parentheses, saves and temporarily removes all locals, except those listed and their descendants.

```
>N
```

```
>N DICKENS,ELIOT
>N (TOLSTOY,IBSEN)
```

OPEN

OPEN*:postcondition* Device*:OpenParameters, . . .*

Open Parameters =>

DeviceParameters*:timeout:MnemonicSpecs*

Device Parameters =>

Device Parameter

(Device Parameter*:Device Parameter: . . .*)

Device Parameter =>

vendor-specific expression

Device Keyword

Device Attribute=expression

Mnemonic Specs =>

Mnemonic Space

(Mnemonic Space*,Mnemonic Space, . . .*)

Gets ownership of an I/O device, selects the list of available mnemonic spaces for that device, and sets the current mnemonic space to the first in the list selected. OPEN can also use device parameters to manipulate the device as it is acquired. Trying to open a device with a mnemonic space it doesn't support causes error M35; trying to open it with incompatible mnemonic spaces causes error M36. Timed OPEN sets $TEST to whether it got ownership in the specified time. [*Implementation-specific OPEN parameters and nonstandard mnemonic spaces are not portable.*]

```
>O LOGFILE,PRINTER::60 E  D FAIL Q
>O:AUTHOR="DICKINSON" "MIND":("NEW"):1
>O DISPLAY:::"X3.64"
```

QUIT

QUIT*:postcondition*

QUIT*:postcondition* Expression

Ends the current process level and returns a value; doing so when an argument is not expected causes error M16. Argumentless QUIT ends the current process level without returning a value, but if one is expected it causes error M17. QUIT can also be used to end a FOR loop on the same line.

```
>Q
>Q "FREUD"
>S AUTHOR="PROUST",NAME="AUTHOR" Q @NAME
```

READ

READ*:postcondition* ReadArgument*, . . .*

ReadArgument =>

String Literal

Formatting String

Variable:*timeout*

Variable*#ReadCount*:*timeout*

Formatting String =>

FeedsTab

/ControlMnemonic*(expression,expression, . . .)*

Feeds => !s (line feeds) or #s (form feeds)

Tab => ? followed by column number

Variable =>

local, global, or structured system variable

ReadCount => integer expression

Gets input from the current I/O device and puts the response in the specified variables. Any text and format control characters in the argument of the READ command are output on the current device. Timed READ sets $TEST to whether READ got a response in the specified time.

When the argument contains an asterisk preceding a variable name, a code representing a single character is obtained. When the argument contains a variable followed by

a "#" and a numeric expression, this expression specifies the maximum number of characters to accept. A number less than zero causes error M18. [*The codes returned by READ * are defined by the MUMPS implementor and not portable. Use of nonstandard control mnemonics is not portable.*]

```
>R #!!,AUTHOR,!?5,WORK
>R *CODE
>R !,NAME#10
>R "NAME:",AUTHOR G PORTRAIT:AUTHOR="JOYCE"
>R !?10,"WOOLF?",YES:30 I '$T W !,"AFRAID?"
```

SET

SET*:postcondition* SetDestination=Expression*, . . .*

Set Destination => SetLeft or (SetLeft,*SetLeft, . . .*)

Set Left =>

local, global, or structured system variable

$DEVICE or $KEY or $X or $Y

$ECODE or $ETRAP

$EXTRACT(string,*from,to*)

$PIECE(string,delimiter*,from,to*)

Puts values into variables. When the SET destination is a list of destinations within parentheses, each destination is given the value following the assignment symbol. The $EXTRACT and $PIECE destinations change the specified part of their first arguments. Trying to set $X or $Y to a negative or non-integer value causes error M43.

```
>S AUTHOR="KAFKA",STORY="METAMORPHOSIS"
>S AUTHOR="BORGES",^CANON(AUTHOR)=1
>S (AUTHOR,CURRENT,^CANON("ACTIVE"))="NERUDA"
>S $ET="TRAP^CANON",$EC=",U42,"
>S $P(^CANON("PESSOA"),"^",3)="MENSAGEM"
>S $E(^CANON("CRITIC"),1,20)=$J("BLOOM",20)
```

TCOMMIT

TCOMMIT:*postcondition*

Commits and ends the current transaction: makes its global changes visible. A TCOMMIT when there is no current transaction causes error M44.

```
>TC:AUTHENTC
```

TRESTART

TRESTART:*postcondition*

Rolls back the current transaction (see TROLLBACK), optionally restores some or all of the symbol table (as dictated by the TSTART command, see below), and starts the current transaction again. Attempting to restart a non-restartable transaction rolls back the transaction, ends it, and causes error M27. A TRESTART when there is no current transaction causes error M44.

```
>TRE:MISSING
```

TROLLBACK

TROLLBACK:*postcondition*

Rolls back a transaction; that is, undoes its global changes and releases any locks acquired within the transaction. A TROLLBACK when there is no current transaction causes error M44.

```
>TRO:CANCEL
```

TSTART

TSTART:*postcondition*

TSTART:*postcondition Variables:TransParameters*

Variables =>

local variable name

(local variable name,*local variable name, . . .*)

* or ()

Transaction Parameters =>

Parameter

(Parameter:*Parameter: . . .*)

Parameter =>

SERIAL

TRANSACTIONID=expression

Z . . . =expression

Starts a restartable transaction. Argumentless TSTART starts a nonrestartable transaction. When variables are specified, they are restored to their previous values if the transaction is restarted. [*Use of transaction parameters starting with Z is not portable.*]

```
>L ^CANON("UPDATE") TS
>TS:OK :T="BECKETT"
>TS (CHANGES,VALUES):(S:T="HOMER")
```

USE

USE:*postcond* Device:*DeviceParams:MnemSpace, . . .*

Device Parameters =>

Device Parameter

(Device Parameter:*Device Parameter: . . .*)

Device Parameter =>

vendor-specific expression

Device Keyword

Device Attribute=expression

Picks the current device from the list of I/O devices owned by the current process, and the device's mnemonic space from the list currently available for that device. USE can also use device parameters to manipulate the current device as it is selected. [*Implementation-specific USE parameters and nonstandard mnemonic spaces are not portable.*]

```
>U:AUTHOR="HERODOTUS" "LOG.TXT"
>U:AUTHOR="THUCYDIDES" HISTORY::"X3.64"
```

VIEW

VIEW:*postcondition* ViewArgument

Returns or changes implementation-dependent information. [*Arguments and behavior of VIEW are nonstandard.*]

```
>VIEW:AUTHOR="PLATO" MEMORY
```

WRITE

WRITE:*postcondition* WriteArgument*, . . .*

WriteArgument =>

Expression

Formatting String

*Character Code

Formatting String =>

FeedsTab

/ControlMnemonic*(expression,expression, . . .)*

Feeds => !s (line feeds) or #s (form feeds)

Tab => ? followed by column number

Variable =>

local, global, or structured system variable

Formats and outputs values to the current I/O device. When an argument includes an asterisk followed by an integer value, one character whose code (not necessarily ASCII) is the number represented by the integer is sent to the current device; the effect this code has on the device is implementation-specific. [*The codes used by WRITE * are defined by the MUMPS implementor and not portable. Use of nonstandard control mnemonics is not portable.*]

```
>W "SOPHOCLES"
>W #!!?15,"EURIPIDES: ",PLAY
>W *7
>W !,^CANON("ACTIVE"),?30,WORK
```

XECUTE

XECUTE*:postcondition* Expression*:postcondition, . . .*

Interprets and executes a value as MUMPS text. XECUTE provides a means of interpreting a data value created during program execution as if it were MUMPS code. Each argument of the XECUTE command is interpreted as if it were the text part of a line of MUMPS code (without label, line start indicator, or line level indicator).

```
>X "S AUTHOR=""ARISTOTLE"""
```

Z . . .

All implementation-specific command names begin with Z. [*Commands whose names begin with Z are not portable.*]

For more information on commands, see chapter 8 of *MUMPS 1995*.

9 · CHARACTER SET PROFILES

MUMPS systems can be assigned a default character set profile; individual processes, globals, devices, and routines can be assigned their own character sets. Structured system variables define these defaults and assignments.

9.1 · CHARACTER SETS ASCII & M

ASCII and M operate upon the same set of characters—the seven-bit ASCII character set—and use the same pattern code definitions, but differ in their collation algorithms.

ASCII collates in strict ASCII numeric order applied lexically. That is, 1 collates before 2, but so do 12 and "1A", just as "A" collates before "AB" which collates before "B".

M collates the same except that canonical numbers collate numerically ahead of all other strings. To use the previous example, "A", "AB", and "B" will collate the same as they do in the ASCII character set, but 2 collates ahead of 12, and all three numbers collate before the string "1A".

9.1.1 · TABLE OF ASCII/M CHARACTERS

#	CHAR	CODES	#	CHAR	CODES	#	CHAR	CODES
0	NUL	C,E	11	VT	C,E	22	SYN	C,E
1	SOH	C,E	12	FF	C,E	23	ETB	C,E
2	STX	C,E	13	CR	C,E	24	CAN	C,E
3	ETX	C,E	14	SO	C,E	25	EM	C,E
4	EOT	C,E	15	SI	C,E	26	SUB	C,E
5	ENQ	C,E	16	DLE	C,E	27	ESC	C,E
6	ACK	C,E	17	DC1	C,E	28	FS	C,E
7	BELL	C,E	18	DC2	C,E	29	GS	C,E
8	BS	C,E	19	DC3	C,E	30	RS	C,E
9	HT	C,E	20	DC4	C,E	31	US	C,E
10	LF	C,E	21	NAK	C,E	32	SP	P,E

#	CHAR	CODES	#	CHAR	CODES	#	CHAR	CODES
33	!	P,E	65	A	A,U,E	97	a	A,L,E
34	"	P,E	66	B	A,U,E	98	b	A,L,E
35	#	P,E	67	C	A,U,E	99	c	A,L,E
36	$	P,E	68	D	A,U,E	100	d	A,L,E
37	%	P,E	69	E	A,U,E	101	e	A,L,E
38	&	P,E	70	F	A,U,E	102	f	A,L,E
39	'	P,E	71	G	A,U,E	103	g	A,L,E
40	(	P,E	72	H	A,U,E	104	h	A,L,E
41	)	P,E	73	I	A,U,E	105	i	A,L,E
42	*	P,E	74	J	A,U,E	106	j	A,L,E
43	+	P,E	75	K	A,U,E	107	k	A,L,E
44	,	P,E	76	L	A,U,E	108	l	A,L,E
45	-	P,E	77	M	A,U,E	109	m	A,L,E
46	.	P,E	78	N	A,U,E	110	n	A,L,E
47	/	P,E	79	O	A,U,E	111	o	A,L,E
48	0	N,E	80	P	A,U,E	112	p	A,L,E
49	1	N,E	81	Q	A,U,E	113	q	A,L,E
50	2	N,E	82	R	A,U,E	114	r	A,L,E
51	3	N,E	83	S	A,U,E	115	s	A,L,E
52	4	N,E	84	T	A,U,E	116	t	A,L,E
53	5	N,E	85	U	A,U,E	117	u	A,L,E
54	6	N,E	86	V	A,U,E	118	v	A,L,E
55	7	N,E	87	W	A,U,E	119	w	A,L,E
56	8	N,E	88	X	A,U,E	120	x	A,L,E
57	9	N,E	89	Y	A,U,E	121	y	A,L,E
58	:	P,E	90	Z	A,U,E	122	z	A,L,E
59	;	P,E	91	[	P,E	123	{	P,E
60	<	P,E	92	\	P,E	124	|	P,E
61	=	P,E	93	]	P,E	125	}	P,E
62	>	P,E	94	^	P,E	126	~	P,E
63	?	P,E	95	_	P,E	127	DEL	C,E
64	@	P,E	96	`	P,E			

9.2 · CHARACTER SET JIS90

JIS90, a character set for representing Japanese characters, is described in two separate standards: *Japanese Industrial Standard JIS X0201-1990 8-bit coded character sets for information interchange,* and *Japanese Industrial Standard JIS X0208-1990 8-bit double byte coded KANJI sets for information interchange.*

MUMPS 1995 Annex H describes the relationship between these two standards within JIS90, their pattern codes and collation, and which characters may be included within MUMPS names (variables, routines, labels, etc.).

COLOPHON

The first printing of this book was designed and set into type in 1995 in Seattle by the author in Bitstream's ITC Friz Quadrata (Flareserif 861) and Monotype's Courier New, using Microsoft Word version 6.0a on Intel 386 and 486 machines.

The author reset the second printing in 2010 using Neooffice 3.0 on a Macintosh Macbook Pro; his goals were (1) to update the software format, (2) to convert its styles to create a navigation outline for the portable-document format, and (3) to update the layout and design to be more readable, restful on the eyes, and typographically coherent.

The text face is Friz Quadrata version 2.00. Swiss designer Ernst Friz created it in 1965 as a solo Roman face for Visual Graphics Corporation. New York designer Victor Caruso added the bold face for International Typeface Corporation's (ITC) 1973 release. French designer Thierry Puyfoulhoux added the italic and bold italic faces for ITC in 1992. Friz Quadrata is a spur-serif typeface legible enough for setting extended text; its clean lines and precision suit it to formal engineering texts like this pocket guide, but its open counters help breathe life and character into the text. The English text is set 8/14.

The code face is Courier 10 Pitch (Fixed Pitch 810) version 1.01, developed by Howard "Bud" Kettler in Lexington in 1955 for IBM typewriters and digitized by Bitstream in the 1980s. Designed to give dignity to official documents, Courier has become the most famous monospaced typeface; most software is printed in Courier. Courier 10 Pitch's color blends better with text faces than Courier New (which is far too light). The MUMPS text is set 8/14.

About the author

Frederick D. S. Marshall founded the VISTA Expertise Network, cofounded WorldVistA, is vice-chair of the MUMPS Development Committee, and is a tier-five, master-hardhat VISTA developer who has implemented, programmed, written about, taught about, and strategized about VISTA for twenty-six years.

AN IMPRINT OF
THE VISTA EXPERTISE NETWORK
819 NORTH 49TH STREET, SUITE 203
SEATTLE, WASHINGTON 98103-6576
INFO@VISTAEXPERTISE.NET
WWW.VISTAEXPERTISE.NET
(206) 632-0166

VISTA
EXPERTISE NETWORK