programming language

MUMPS

digital

146 MAIN ST. MAYNARD, MASS.

DEC-15-GXZC-D

M U M P S

Programming Language

First printing January 1971
Second printing October 1971
Revised April 1972

For additional copies, order No. DEC-15-GXZC-D from Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts, 01754
Price $4.00

FOREWORD

MUMPS (Massachusetts General Hospital Utility Multi-Programming System) is a single-language interactive time sharing system developed for a medium scale computer.

This language was developed by the Laboratory of Computer Science of Massachusetts General Hospital and Harvard Medical School, Boston, Massachusetts. This development was supported by research grants from the National Center for Health Services Research and Development (HS 00240) and from the National Institute of General Medical Sciences (GM 15287). This manual is adapted from a users' manual printed by the Laboratory of Computer Science.

This edition of the MUMPS Programming Language manual is a revision of manual number DEC-15-GXZB-D. Technical changes in the program as documented have been marked with a solid line; editorial changes have been marked with a broken line. Examples:

Pages changed: ii, 1-2, 4-5, 4-6, 4-10, Chapter 5 all pages, 6-3, 7-2, A-1, Readers Comments page. The Index has been updated accordingly.

CONTENTS

INTRODUCTION

MUMPS (the Massachusetts General Hospital Utility Multi-Programming System) is a user-oriented, general-purpose programming language integrated with an interactive time-sharing system. The MUMPS language is user-oriented in that it is a high-level language that is easy to learn and use. The MUMPS environment allows a programming session to center around a reactive terminal, thus minimizing the user's time in programming a problem, the computer's time needed in checking it out, and, most important, the elapsed time required to obtain a final running application program. The system is especially powerful in on-line data management applications requiring interactive terminals structured to provide a general information system. The MUMPS language has the capability to manipulate character strings and data files with the same ease and flexibility that it handles numeric and Boolean expressions.

The hardware system is composed of three basic elements -

1) A medium scale central processor.
2) Scanners capable of interfacing to local or remote Teletype[1] like terminal devices.
3) At least one high-speed disk memory system.

MUMPS was originally conceived as a special purpose problem-oriented system for medical information processing at Massachusetts General Hospital. It was soon realized that the general-purpose character of the system indicated that it could serve as the foundation of a clinical data management system. The general objective of using the system for the acquisition, storage, and retrieval of medical record information gained impetus as a growing number of particular application areas were integrated into the structure.

The MUMPS system functions in a variety of on-line clinical applications - automated interpretive patient histories, laboratory information processing, medical diagnosis, entries of physical examination notes and radiology reports, medical records, hospital care health planning, critical data management, and others. As the system evolved, it became clear that it could provide a reliable service function in a wide range of applications areas.

[1]Teletype is the registered trademark of the Teletype Corporation.

Unlike other on-line information systems, MUMPS language programs are not compiled into machine language, but are executed by an interpreter that is resident within the time-sharing system. This approach not only facilitates the development of programs in an interactive environment, but also permits implementation of several novel features in the language.

In the first three chapters, the reader is introduced to the basic structure and components of the MUMPS language. These chapters are structured to enable the reader to readily acquire a working knowledge of the language. Important points are occasionally repeated for emphasis, and examples are abundant. Most examples are self-contained programs that the reader can actually type in and run while learning the language, thus reinforcing the concepts that are introduced.

Learning MUMPS at a Terminal

The best way to learn the MUMPS language is at a terminal connected to a MUMPS system. The command and program examples in the following chapters can be typed in and executed. If you are in doubt about the effect of a command or a form of syntax, try it out to see how MUMPS interprets it.

The status of every terminal is set up beforehand by the MUMPS system manager, through the MUPAK utility package. Usually, a practice terminal is set up in the "scratchpad" mode of operation, which permits you to type in commands for direct execution, compose program steps, and execute programs in "indirect" mode. A limited storage area ("user partition") is reserved for your program steps and variable data. Access to other programs or the MUMPS global data base is usually restricted, and you cannot file your programs permanently.

A typical log-in procedure for a scratchpad user is as follows (check with your MUMPS system manager to obtain your user name, scratchpad key, and the current procedures):

1. Press the BREAK key. MUMPS responds by typing:

   ```
   MUMPS LINE 1
   ID
   ```

2. Type your user name, scratchpad key [in brackets], and an asterisk:

   ```
   MUMPS LINE 1
   ID STE[3.2]*
   ```

   (User inputs are underlined)

   If the scratchpad key is 0.00, it and the brackets are omitted (close up the space).

3. Press the ENTER key. (The ENTER key may also be labeled ESC, ALT MODE, or PREFIX, depending on the type of terminal.) If your ID codes are valid, MUMPS responds with a CRLF (carriage return-line feed) and types a right caret:

```
    MUMPS LINE 1
    ID STE[3.2]*
    >
```

The right caret (>) indicates that you are communicating with MUMPS in the "direct" mode of operation. You can type in any of the command examples or program steps that appear in the following chapters, or compose and execute your own programs. It is possible to become stuck accidentally in the "indirect" mode of operation (executing numbered program steps) so that you are unable to enter more commands from the keyboard. If MUMPS fails to type the right caret after you enter a command, you are probably in that condition. To regain control, press the BREAK key. MUMPS prints out an error message and returns to direct mode.

To terminate a session at the terminal, enter a HALT command (Chapter 4) or press the ENTER key after the terminal has requested your ID. MUMPS types out THANK YOU to indicate the end of the session.

Fully detailed sign-on, sign-off, and file protection descriptions appear in Chapter 7. Once you are acquainted with the advanced features of the MUMPS language, you can generate and file your own programs, use these and the programs of other users, and use and modify the MUMPS global data arrays.

CHAPTER 1

BASIC LANGUAGE FEATURES

1.1 ARITHMETIC OPERATORS

MUMPS allows four arithmetic operators. They are:

+	addition
-	subtraction
*	multiplication
/	division

Any "well-formed" combination of these operators and numerical operands constitutes a number-valued expression (NVE) (see 1.4).

1.2 OUTPUT - THE TYPE COMMAND

MUMPS can be used to evaluate and TYPE the answer to an arithmetic expression. To use the TYPE command, type "TYPE", followed by one space, followed by the numeric expression desired, and then press the ENTER key. MUMPS evaluates the expression and types out the result. For example:

```
>TYPE 1+(2*3)-4/10
6.60
>
```

The system types a carriage return and line feed followed by a right caret (>) to indicate that it has returned control to the user and is waiting for the next command to be ENTERed. In the example above and in other examples throughout this manual, all input typed by the user is underlined; system output is not underlined.

1.3 RECOVERY FROM TYPOGRAPHICAL ERRORS

Occasionally, when entering data or writing a MUMPS statement, the user may make a typographical error. There are two ways to correct errors of this sort: character rubout or line deletion.

One or more characters in the line may be deleted by striking the RUBOUT key. This causes the system to type out a backslash (\ ) and to delete the last non-deleted character from the line. The user may then continue entering text. For example:

```
>TYKDY\\\PE 1+3*4
13
>
```

MUMPS recognizes the above statement as "TYPE 1+3*4" and executes it correctly.

Every character typed in since the last ENTER may be deleted by typing Control U[1]. The system responds by echoing up-arrow U (↑U) followed by a carriage return and line feed, and then waits for the correct text to be typed in.

1.4 NUMBERS IN MUMPS

MUMPS calculations are performed in fixed-point arithmetic and numbers retain an accuracy of TWO decimal places. Results of calculations that produce more than two decimal places are truncated (as opposed to being rounded). The largest number value allowed in MUMPS is 1310.71, positive or negative. It is important to note that these restrictions apply to intermediate as well as to final results; if the magnitude of any result exceeds 1310.71, MUMPS detects the error, prints the error message MAXIM, and enters direct mode.[2]

For example:

```
>TYPE 100*100/99

?  MAXIM
>
```

The intermediate result (10,000) is too large.

```
>TYPE 2/3*3
1.98
>
```

The intermediate result (0.66) was truncated to 2 decimal places.

```
>TYPE 1.3*1.117

?  MINIM
>
```

The second number has too many decimal places. Numbers in MUMPS are limited to two places.

[1]Control U (CTRL U) is formed by pressing the CTRL key while striking the letter U.
[2]The range of MUMPS numbers can be extended by the $MONEY function described in Chapter 5.

It is vital that the programmer understand the limitations of MUMPS arithmetic; the truncation of results (as in the second example above) is the cause of many logical errors in MUMPS programs.

NOTE

NO ERROR MESSAGE IS OUTPUT WHEN A TRUNCATION IS PERFORMED.

1.5 PRECEDENCE OF ARITHMETIC OPERATORS

MUMPS evaluates arithmetic expressions from left to right, with two exceptions. First, multiplications or divisions occurring immediately to the right of an addition or subtraction are performed prior to the addition or subtraction. (Note that multiplication following division is performed after the division, in the normal left-to-right order.) Secondly, the user may change the order of evaluation by the use of parentheses. For example:

EXPRESSION	RESULT
1+3*4/12	2
(1+3)*4/12	1.33
1+3*(4/12)	1.99
(1+3)*(4/12)	1.32
(1+3*4)/12	1.Ø8
3+2/4+1	4.5Ø
(3+2)/4+1	2.25
3+2/(4+1)	3.4Ø
3+2*4+1	12
(3+2)*4+1	21
3+2*(4+1)	13
1/2*3/4	Ø.37
1/(2*3)/4	Ø.Ø4
1/(2*3/4)	Ø.66
1*2/3*4	2.64
1*2/(3*4)	Ø.16

The subtraction operator (-) may also be used for arithmetic negation (i.e., it may be used as a "unary minus"). For example:

```
>TYPE -(3+4)
-7
>
```

1.6 VARIABLES IN MUMPS - THE SET AND KILL COMMANDS

MUMPS can assign values to variables by means of the SET command. The user can specify the name of a variable by using from one to three letters. More than three letters may be used, but those after the third are ignored. Thus VAR, VARIABLE, VARIANCE, and VARIANT are all recognized by MUMPS as VAR. % is legitimate as the first character of a variable, but certain % variables are predefined and have a special meaning (see Table 4-1).

To use the SET command, type "SET", followed by a space, followed by the variable name to be set, followed by an equal sign (=), followed by the expression whose value is to be assigned to the variable name, and then press the ENTER key. MUMPS evaluates the expression and, if legal, assigns the result to the variable name. For example:

```
>SET A=1+3*5/3
```

SET may also reference values previously assigned to a variable. For example:

```
>SET B=A*3+12
```

A single SET command may be used to SET several variables:

```
>SET A=1,B=4+A,D=A+B,C=D-B
```

As can be seen, arguments (e.g., A=1) are separated by a single comma (,). Once a variable has been defined, it may be used to define other variables, as above, or its value may be typed out using the TYPE command.

When used with the SET command, the equal symbol (=) does not mean "equality" in the usual mathematical sense. Rather, its meaning is more akin to "assign this value to". For example, once defined, a variable may be set to the value of an expression containing itself:

```
>SET A=23

>TYPE A
23
>SET A=A+3

>TYPE A
26
>
```

The value of a variable may be changed at any time by giving it a new value with a SET command:

```
>SET A=4

>TYPE A
4
>SET A=23

>TYPE A
23
>
```

When a variable is given a new value, MUMPS "forgets" the old one and assigns the new one to it. Often, it is desirable to eliminate a variable completely rather than give it a new value. This may be done by the KILL command:

```
>KILL A

>TYPE A

?  UNDEF
>
```

Here MUMPS was made to "forget" the existence of the variable A by execution of the "KILL A" command. Like the SET command, KILL may take a list of arguments. When the command is executed, all the variables in the list are deleted or "forgotten".

```
>KILL A,F,ABC,SUM

>
```

Variables which are not defined may be KILLed without causing an error.

If no arguments follow a KILL command, all locally defined variables (i.e., those defined within your user partition) are deleted. In like manner, if no arguments follow a TYPE command, all locally defined variables are typed.

1.6.1 Subscripts

Variables in MUMPS may be subscripted. Subscripts can be used to indicate different "cases" or "examples" of a variable, or different positions in a matrix, etc. To use subscripts in any expression, simply type the name of the variable, followed by an open parenthesis, followed by the expression whose numerical value is to be taken as the subscript, and a close parenthesis. Subscripted variables must always have only ONE level of subscripting, and the value of the subscript must be a positive number from 0.00 to 327.67. Subscripts themselves may contain variables, provided any such variables are assigned numerical values. Subscripted and nonsubscripted variables may not use the same name, e.g., N and N(1) cannot be defined at the same time.

The following are all legal subscripted variables:

```
AGE(2)          ABC(25.08)
AGE(Q)          ABC(N)
AGE(O+3*B)      ABC(4+(B+2*N))
```

Subscripts can themselves be subscripted variables:

```
ABC(ABC(AGE(3)))        NUM(AGE(I+4))
```

An entire array may be deleted by killing the array name.

```
>KILL ABC
>
```

An individual array member may also be killed.

```
>KILL ABC(25.Ø8),ABC(N)
```

1.7 USE OF THE FOR CLAUSE

Often it is desirable to repeat the same calculation over and over again with the only difference being the value of some number in the calculation. The use of the FOR clause allows the useful tool of iteration to be implemented with a minimum of program space. Essentially, the FOR clause causes the command(s) following it on the same line to be executed FOR a specified set of values of the same variable.

```
>FOR I=1,2,3,4,8,34 SET X(I)=I+10

>TYPE X(4)
14
>


>FOR I=1:1:9 TYPE 2*I," "
2 4 6 8 10 12 14 16 18
>
```

In the first example above, the FOR clause sets I=1 (the first number in the FOR list) and then executes the command following the FOR clause. It then repeats the execution of the SET command for the other values of I as specified in the FOR list.

In the second example, the FOR clause sets I=1, the first number in the FOR list specification, and tests the value of I (INDEX) against the third number (TERMINATOR) in the FOR list (9). Since the value of the INDEX variable I is NOT LARGER than the value of the TERMINATOR (9), the command following the FOR clause is executed. Next, FOR increments the INDEX by the value of the INCREMENT (1) (between the colons), and again tests the value of the INDEX to see if it exceeds the TERMINATOR. The sequence is repeated until the value of the INDEX is greater than the value of the TERMINATOR.

The two types of FOR lists may be combined in the same FOR clause. The following example illustrates such a combination:

```
>FOR I=5,8,33:6:57 TYPE I+I/3,!
6.66
10.66
44
52
60
68
76
```

The STARTING VALUE, INCREMENT, and TERMINATOR may be numbers, arithmetic expressions, or numerically defined variables:

```
>SET A=5

>FOR I=1+1/2:A:33.40 TYPE I," "
1.50 6.50 11.50 16.50 21.50 26.50 31.50
>
```

1.8 FORMAT CONTROL

Often it is necessary to present large amounts of data in some well-formatted manner. MUMPS allows three special format characters:

#	page-feed (or form-feed)
!	carriage-return line-feed (CRLF)
?N	tabulate (start typing N spaces from left margin)

These characters, as well as quoted text literals, may appear in a TYPE statement (as well as in certain other statements discussed later). When MUMPS scans the TYPE command's list of arguments, if it finds one of the above format characters, it outputs the appropriate formatting operation as well as the quoted text string.

```
>FOR A=1,2,4:4:12 TYPE "VALUE=",A*A,!
VALUE=1
VALUE=4
VALUE=16
VALUE=64
VALUE=144

>
```

In the above example, note that a comma is used following "A*A" and before the CRLF character (!). Commas that are normally used as delimiters in an argument string may not be omitted preceding and/or following format control characters. (However, !! is permitted.)

The tabulation character (?N) is particularly useful for preparing columns of data:

```
>FOR R=Ø.5:Ø.1:1.2 TYPE "R=",R,?10,"A=",3.14*R*R,!
R=Ø.5Ø    A=Ø.78
R=Ø.6Ø    A=1.12
R=Ø.7Ø    A=1.53
R=Ø.8Ø    A=2
R=Ø.9Ø    A=2.53
R=1       A=3.14
R=1.1Ø    A=3.79
R=1.2Ø    A=4.51

>
```

In the above example, the format expression ?10 causes the "A=" column to start on the 10th character from the absolute left margin (i.e., the margin that is in effect when %M=Ø). If %M and ?N conflict, %M takes precedence. For example, if %M is set to 10 and a piece of text is formatted to start at position ?8, the text will start at character position 10 from the absolute left margin.

In any line of text, if one text string overlaps the starting position for a ?N-formatted string, the ?N string starts on the next available character position. For example:

```
>SET %M=Ø

>SET A=17

>TYPE A,?6,"METERS"
17    METERS

>SET A="SEVENTEEN"

>TYPE A,?6,"METERS"
SEVENTEENMETERS
>
```

CHAPTER 2

WRITING MUMPS PROGRAMS

2.1 STEPS AND PARTS

The simple commands presented as examples in Chapter 1 are considered "direct" steps -- that is, they are executed by MUMPS as soon as they are entered. You can see the result of a direct command immediately, but a direct command must be retyped every time it is performed. The real power of MUMPS lies in the automatic sequential execution of commands stored in memory. Commands executed in this manner are considered "indirect" steps.

2.1.1 Steps

Indirect steps are composed and entered while you are still in direct mode (identified by the right caret that MUMPS types after each ENTER). To compose a step, type a positive two-place decimal number followed by a space and the command string that you wish to have "remembered":

```
>1.10 TYPE A*31.2
>
```

The fractional part of the step number must be non-zero (i.e., "1.00" is not legal).

After typing the command string, press the ENTER key (also known as ESC, ALT MODE, or PREFIX). MUMPS does not execute the command at this time; it stores the step in its memory and responds by typing a CRLF followed by a right caret. MUMPS is then ready for entry of another step or a direct command.

The STEP NUMBER is the label by which the step may be referenced in other sections of the program. Steps are stored in step number order. If you wish to change the contents of a STEP after you have ENTERed it, you may simply retype it, using the same STEP number. MUMPS replaces the old STEP with the new one having the same step number. Similarly, if you have defined STEPS 2.34 and 2.50, and now wish to INSERT a new step between the two old ones, you may do so by using a step number between 2.34 and 2.50. The new step will be inserted in the program memory between the old STEPS 2.34 and 2.50.

2.1.2 Parts

MUMPS also handles groups of steps as PARTS. The PART that a STEP belongs to is indicated by the integer portion of the step number. For example, step 2.35 is one of the steps in PART 2. PART numbers may range from 0 to 999. Within a PART, steps may take decimal values of from .01 to .99. By convention, MUMPS programmers write their own programs so that a PART contains all the coding necessary for a specific task. Thus, if a program were designed to perform many different tasks, each PART in the program would be written to perform just one of those tasks.

2.2 THE DO STATEMENT

The DO statement causes MUMPS to execute indirect parts or steps automatically in step number sequence. Consider the following short program to calculate the area of circles of radius R:

```
1.1Ø TYPE "R=",R,?1Ø,"A="
1.15 SET A=3.14*R*R
1.2Ø TYPE A,!
```

The three STEPS (1.10, 1.15, and 1.20) are stored in numerical order by MUMPS. Together, they form a program that has a single part (PART 1). To start the program, load the variable R with a numerical value and use a DO command:

```
>SET R=1.25

>DO 1
R=1.25     A=4.9Ø

>SET R=1+3/8

>DO 1
R=1.37     A=5.89
```

In executing the DO command, MUMPS "does" each step, in numerical order, in the part referred to by the DO command. When there are no more steps in PART 1, MUMPS does a CRLF and types the right caret, indicating that it is ready to accept another command string or store a new step.

The DO command, like the TYPE command, may be modified by one or more FOR statements. For example, you could use the program of STEP 1 to type the areas corresponding to several radius values by entering the following statement:

```
>FOR R=1.00:0.25:2.50 DO 1
R=1         A=3.14
R=1.25      A=4.90
R=1.50      A=7.06
R=1.75      A=9.60
R=2         A=12.56
R=2.25      A=15.88
R=2.50      A=19.62
```

Remember, a FOR clause causes repeated execution of whatever command string follows it on the same line. Above, the command string following the FOR is a DO statement, and thus all of PART 1 is executed for each value of R, the INDEX of the FOR clause.

Note that although STEP 1.15 requires a value for the variable R for successful execution, the variable need not be defined when the step is typed in. In MUMPS, variables need not be defined with a value until the time they are actually referenced or used. In this example, the initiating FOR clause assigns a new value to R for every repetition of PART 1.

DO statements may also be stored as command strings within steps of a MUMPS program. For example, the following program (PART 2) makes reference to a single step of PART 1:

```
2.15 SET R=2.3
2.20 DO 1.15
2.25 TYPE "AREA=",A,!

>DO 2
AREA=16.60
```

A DO statement could reference all of PART 1 as well as a single step:

```
3.10 SET L=15.25
3.15 SET R=2.75
3.20 DO 1
3.25 TYPE "VOLUME=",A*L,!

>DO 3
R=2.75    A=23.73
VOLUME=361.88

>
```

A DO statement that references another part may also be controlled by a FOR clause, so that the other part is executed repeatedly, as in the following program.

```
4.10 SET SUM=A+B,DIF=A-B
4.39 TYPE "SUM= ",SUM
4.40 TYPE " DIFFERENCE= ",DIF,!


6.10 SET A=10
6.30 FOR B=2,5,15 DO 4


>DO 6
SUM= 12 DIFFERENCE= 8
SUM= 15 DIFFERENCE= 5
SUM= 25 DIFFERENCE= -5

>
```

MUMPS begins this program at STEP 6.10, the first step in PART 6. When it encounters the FOR and the DO statements in STEP 6.30, it "does" each step in part 4 for the three values of B. After PART 4 has been "done" for each value of B, MUMPS does not find any more steps in PART 6 (called by the DO 6 command string in direct mode) and so it types out the right caret and waits for the next command.

In computer terminology, the DO command allows MUMPS to execute a PART of a program written as though it were a subroutine within the main program. Usually, subroutines are written so that many different sections of the program may cause the execution of the subroutine at different times and with different sets of data. In a statistical program, for example, PART 34 may be the subroutine for computing the standard deviation of N numbers about an arithmetic mean of MU.

A single DO command may refer to a LIST of steps or parts. For example, provided that the user has previously defined all of the steps and parts referred to below, all of the following statements would be legal DO commands.

```
DO 2,3,67,56.14
DO 1.34
DO 3.45,5,9
FOR I=1:2:46 DO 3,15
FOR ABC=34,56,104,200:1:250 DO 3.45,6
```

2.3 THE IF STATEMENT

In executing a program, MUMPS can make decisions as to what action to take depending on the current value of variables or expressions within the program. An IF clause specifies the condition for which the command string following it on the same line is to be executed. Changing the example of sums and differences above, we can write a program to only type out the differences of the two numbers A and B if they are positive. If the difference is less than zero, the program types Ø.

```
4.1Ø SET SUM=A+B,DIF=A-B
4.2Ø TYPE "SUM= ",SUM
4.24 IF DIF<Ø SET DIF=Ø
4.26 TYPE " DIFF= ",DIF,!


6.28 FOR A=2Ø FOR B=1Ø,18,34,5Ø DO 4

>DO 6
SUM= 3Ø DIFF= 1Ø
SUM= 38 DIFF= 2
SUM= 54 DIFF= Ø
SUM= 7Ø DIFF= Ø

>
```

In this example, the SET command in step 4.24 is executed only when the value of DIF is less than zero. Thus the program makes a decision based on the value of the variable DIF.

2.3.1 Arithmetic Comparison Operators

In general, the decision in an IF clause is made upon examination of a comparison of two expressions, variables, constants, etc. In the example above, the comparison made in step 4.24 is whether or not the value of DIF is less than zero; the specific comparison operator is "less than" (<). The complete set of arithmetic comparison operators in MUMPS is as follows:

	=	equals
	<	less than
	>	greater than
<> or	><	greater than or less than (not equal)
=<	<=	less than or equal
=>	>=	greater than or equal

2.3.2 Boolean Operators

In the example above, the general form of an IF clause is "IF" followed by a comparison sub-clause based on a comparison operator. The power of IF statements may be greatly increased by using another kind of operator, the Boolean Operator, to modify or link together two comparison sub-clauses. In MUMPS the three types of Boolean operators (in order of precedence) are:

'	NOT
&	AND
!	OR (inclusive OR)

For example:

```
IF A=B&C>D DO 2
```

Part 2 is "done" if A equals B, AND C is greater than D; i.e., only if both comparison clauses are TRUE.

```
IF A=B!C>D DO 2
```

Part 2 is done if either or both sub-clauses are true.

```
IF 'A=B DO 2
```

Part 2 is done if A is NOT equal to B. Note that this is equivalent to either of the following statements:

```
IF '(A=B) DO 2

IF A><B DO 2
```

Finally, using the three Boolean operators, many different comparison sub-clauses may be linked together to form complex Boolean expressions:

```
IF A=B!C=3!B=2*(A+B)&'A=C DO 2
```

The decoding of this complex Boolean is left as an exercise to the reader.

2.4 MULTIPLE CLAUSES

IF and FOR clauses may be combined in modifying a MUMPS command string. You may use as many modifying clauses as can fit on one line. THE ORDER OF EVALUATION IS FROM LEFT TO RIGHT.

```
FOR I=1:1:23 IF I*2=B-C FOR J=3,4,9 FOR K=2,3 DO 5,3,7
```

MUMPS takes the following action on this command string. The first (left-most) INDEX variable I is stepped through the specified values. For each value of I, IF the conditions specified in the Boolean sub-clause are met, then J (the second INDEX variable) is stepped through its values. Again for each value of J, the last INDEX variable K is stepped through its values and for each value of K, parts 5, 3, 7 are "done".

2.5 BRANCHING -- THE GOTO COMMAND

In general, MUMPS executes the steps in a part in ascending numerical sequence. One exception to this rule is the DO command which causes the normal sequence of control to be interrupted, transferred for a time to another section of the program, and then returned to continue the normal sequence.

GOTO is another command in MUMPS that alters the normal sequence of execution. In programming terminology, GOTO is a command that transfers control from one area of a program to another. When MUMPS encounters a GOTO command, it "goes" to the beginning of the referenced part or step. Execution then continues in the normal manner. Unlike DO, however, the GOTO command does not imply that control will be returned after execution of the referenced PART or STEP.

For example, let us assume that a program has defined an array of numbers called X(I) for values of I from 1 to 12. The following program first finds the sum of the 12 numbers in PART 1 and then transfers control to PART 2 where the mean is computed and typed out.

```
1.10 SET SUM=0
1.20 FOR I=1:1:12 SET SUM=SUM+X(I)
1.30 TYPE !,"THE SUM EQUALS ",SUM
1.35 GOTO 2

2.10 TYPE ", AND THE MEAN IS ",SUM/12

>DO 1

THE SUM EQUALS 85.80, AND THE MEAN IS 7.15
>
```

In the example above, there could have been another GOTO statement in STEP 2.20 that transferred control to another PART or STEP. This is perfectly legal. In fact, STEP 2.20, if it existed, could have been GOTO 1.10, for example, and control would simply cycle through the above sequence over and over again.

The GOTO statement may be modified by an IF clause, producing the effect of conditional branching. Clearly, however, the GOTO statement should not be modified by a FOR clause since the GOTO does not return control. For the same reason, a GOTO may not have a list of STEPs or PARTs to "go" to. Also, GOTO may be used only in a stored command string, and not in direct mode. Thus, you could not start a program from direct mode by typing "GOTO 1".

2.6 THE QUIT COMMAND

In the previous examples, a PART was considered "finished" by MUMPS when there were no more STEPs in the PART. A program may abort execution of sections of a PART or STEP by the use of the QUIT command. For example:

```
4.10 FOR I=1:1:100 IF AGE(I)=91 QUIT
```

In STEP 4.10, the search through array AGE stops when 100 values have been searched, or when an array member having the value 91 has been found. The following program illustrates conditional termination of a PART.

```
17.31 TYPE !,"SUM="
17.35 SET SUM=0
17.40 DO 19

19.20 FOR KI=1:1:50 SET SUM=SUM+AGE(KI)
19.30 IF SUM=0 QUIT
19.40 TYPE SUM," MEAN= ",SUM/50,!

>DO 17

SUM=1275 MEAN= 25.50

>
```

The last step in PART 19 is only executed if the sum of the array AGE is not zero.

2.7 INPUT -- THE ASK COMMAND

In the examples considered thus far, all of the variables used in the expressions have been SET by commands stored within the program. The power and flexibility of MUMPS is greatly increased by allowing the user to define numerical data in an interactive mode with the computer. Let us assume for the moment that a program is needed to form the sum of N arbitrary numbers. The following program will do the job:

```
2.10 SET SUM=0
2.15 ASK "WHAT IS THE NUMBER OF ITEMS? ",N,!!
2.20 TYPE "THANK YOU, LET'S BEGIN:",!
2.30 FOR I=1:1:N DO 3
2.50 TYPE !,"SUM ",SUM

3.10 ASK "ITEM=",ITM(I),!
3.20 SET SUM=SUM+ITM(I)
```

When Part 2 is "done" via the DO command, the program allows the user to define the number (N) of items in his list. It then waits for the user to define each of the items in that list. When N items have been entered by the user (to terminate each item, press the ENTER key) the program types out the sum.

```
>DO 2
WHAT IS THE NUMBER OF ITEMS? 7

THANK YOU, LET'S BEGIN:
ITEM=1
ITEM=4
ITEM=9
ITEM=16
ITEM=25
ITEM=36
ITEM=49

SUM 140
>
```

In the above example, an item might have been expressed as a sum (e.g., ITEM=6+3 rather than 9), since the ASK Command evaluates the expression input. $ROOT(X) is also legal as long as X is defined (for discussion of $ROOT see Chapter 5).

ASK, like TYPE, KILL, and SET, will accept a list of variables to be entered. Like TYPE, it also accepts format characters (line-feed, form-feed, and tab (?N)) and outputs any text strings enclosed in quotes.

```
>ASK ?5,"AGE OF SAMPLE = ",AGE,!,?13," TYPE = ",TYP
    AGE OF SAMPLE = 36
             TYPE = 4
```

2.8 ERROR HANDLING

When MUMPS is unable to process a command for any reason, it outputs a five-character error message. If it is executing a step (rather than a direct command) the step-number precedes the message. The message is followed by CRLF and a caret to indicate that the user has control in direct mode. For example:

```
>TYPE 100*100

?  MAXIM
>

>1.10 YOG
>DO 1.10

? 1.10 CMMND
>
```

Refer to Appendix A for a complete list of MUMPS messages.

2.9 CREATING AND CHANGING A MUMPS PROGRAM

If you type a MUMPS command string preceded by a legal step number (0.01 to 999.99), MUMPS interprets it as a step and stores it in the Step Buffer (an area of your user partition). To change the contents of a step, simply redefine it using the same step number, and MUMPS will update it in the buffer.

To verify the contents of a step or a part or a list of steps and parts, you may use the WRITE command in direct mode:

```
>WRITE 2,3,1.6
```

MUMPS prints out on the terminal all of the steps in PART 2, the single STEP 3.1, and all of the steps in PART 6. When MUMPS finishes writing all of the steps and parts, it returns control to direct mode.

To delete any step or part in the program, you may use the ERASE command. Like WRITE, it accepts a list of steps or parts to be erased.

ERASE and WRITE, WHEN USED WITHOUT A LIST OF STEPS OR PARTS FOLLOWING THEM, REFER TO ALL PROGRAM STEPS IN YOUR USER PARTITION. Be very careful in the use of the ERASE command; know exactly which sections of the program you are deleting.

2.10 CHANGING OR DELETING LOCAL VARIABLES

Local variables are the variables defined during the execution of program steps or direct commands. They are stored in the symbol table portion of your user partition. To verify the contents of any variable, use the TYPE command:

```
>SET I=123.2

>TYPE I
123.20
>
```

Any local variable may be changed by a SET command.

When it is used without a list of variable names as argument, the TYPE command causes your terminal to type out all the variables that are defined in your user partition. In addition, several permanent system variables ($L, etc.) are typed out for your reference. (These are defined in Chapter 4.)

To delete any variable in your partition, use the KILL command. KILL can take a list of variables to be deleted.

KILL and TYPE, WHEN USED WITHOUT A LIST OF VARIABLE NAMES, REFER TO ALL VARIABLES IN YOUR USER PARTITION. Be careful in the use of the KILL command; know exactly which variables you are KILLing. (The special system variables ($L, etc.) are not changed by the KILL command.

2.11 SINGLE-LETTER COMMANDS

In all the examples thus far, the full name was used to specify each command. In actual fact, however, MUMPS makes use of ONLY THE FIRST LETTER of a command word. For example, the statement:

```
FOR I=1:1:10 IF I<QTY FOR ARG=2,4 DO 5
```

can also be expressed as follows:

```
F I=1:1:10 I I<QTY F ARG=2,4 D 5
```

In the above command string, note that the second "I" (standing for IF) is distinguished from the third I (the variable I) by position only. When MUMPS scans a command string, it "remembers" the first letter and ignores all other characters until it finds a space or reaches the end of the line. If it finds a single space and the command requires arguments, MUMPS interprets the characters following the space as arguments.

2.12 MULTIPLE COMMANDS ON A LINE

With the exception of modifying clauses like FOR and IF, the above examples have been restricted to one command per line. In MUMPS, this restriction is unnecessary. The number of MUMPS commands per line is limited only by the 72 characters allotted for a Teletype line. The sequence of execution is as follows: beginning at the first command word on the line, all commands are executed from LEFT to RIGHT, one after the other. When there are no more commands left on the line to execute, MUMPS continues with the next Step in numerical order. Of course, this numerical sequence may be altered by the DO and GOTO commands.

The following example illustrates the use of multiple, single-letter commands within a step.

```
3.10 S A=0 A !"N=",N F I=1:1:N T !,I,"=" A C S A=A+C
3.20 T !"MEAN=" S C=A/N I C<10 T "  ",C Q
3.30 I C<100 T " ",C Q
3.40 T C

>DO 3

N=4
1=4
2=3
3=2
4=1
MEAN=  2.50
```

A double space convention allows a WRITE-all, ERASE-all, KILL-all, or TYPE-all to be followed by other commands in the same step:

```
>1.10 W  T "END OF PROGRAM"
>DO 1

1.10 W  T "END OF PROGRAM"
END OF PROGRAM
>
```

The double space following TYPE is interpreted by MUMPS as "type all". Without the double space, MUMPS might attempt to type the contents of the variable ASK.

2.13 PROGRAM COMMENTS

The MUMPS programmer can insert comments in his program simply by preceding the desired comment with a semicolon (;). However, it should be noted that comments occupy space that could otherwise be used by the program. The following statement illustrates the manner in which the individual lines may be commented.

```
5.10 S AGE=22 ;SET TO SEARCH FOR ALL 22-YR OLDS.

>DO 5.10

>TYPE AGE
22
>
```

When MUMPS encounters such a statement, it considers all characters following the semicolon (up to the end of the line) to be a comment.

2.14 PROGRAM DEBUGGING -- BREAK AND GO COMMANDS

Often, for debugging purposes, a user may wish to interrupt the operation of his program at predetermined points to examine his program data in some detail. After examining the data, he may then wish to resume the normal sequence of operation from the point of interruption. This technique is especially useful in the early stages of program debugging.

The BREAK command interrupts execution of the program, prints out ?, the number of the step where the BREAK command was found, and the word BREAK, and returns control in direct mode. The user has the option of examining local variables as well as other data at this point. He may, for example, change the value of some local array, by executing a statement in direct mode. When the user wishes to resume execution of the program, he issues the command GO in direct mode and presses the ENTER key. MUMPS then continues from where it left off:

```
1.05 TYPE "BREAK EXAMPLE"
1.10 SET I=0
1.15 BREAK
1.20 TYPE "I=",I

>DO 1
BREAK EXAMPLE
?  1.15 BREAK
>TYPE I
0
>GO
I=0
```

If you introduce an error while inspecting results or, for example, you hit BREAK during a TYPE ALL, you lose the breakpoint. MUMPS types an error message and returns you to direct mode. If you attempt to continue with a GO, a STACK error is created:

```
>DO 1
BREAK EXAMPLE
?  1.15 BREAK
>TYPE

D=1.10
I=0
TYP=4
AGE=36
A=SEVENTEEN
SH
?  IOINT                              ← BREAK key pressed
>GO

?  STACK
>
```

CHAPTER 3

STRING HANDLING OPERATIONS

In Chapters 1 and 2 of this manual, you were introduced to the basic features of the MUMPS language as well as some of the commands. In this chapter, we will discuss the STRING capabilities of MUMPS.

3.1 STRING VARIABLES

By now you may have realized that the main purpose of MUMPS is not to perform complex arithmetic with high accuracy. The language has only basic arithmetic operations and its decimal accuracy of two digits is not very impressive when compared to the eight or more place accuracy of scientific languages like FORTRAN. In fact, the main power of MUMPS lies in its ability to perform powerful and flexible operations on STRING data. This capability allows MUMPS to be used with great success in information processing applications.

To begin with, we need a working definition of a string datum. If we consider the set of printing characters that can be typed by a Teletype, we have the universe of elements that can be combined to make a string datum. Exceptions are that @ and \ may not be used in a MUMPS string. For example, consider the following series of Teletype characters:

```
ABCDEFGHI&**%0084-156=./?JG"#*)(,
```

The thirty-four printing characters above constitute a STRING datum. Of course, the particular string used above may make no sense to anybody, but it is still a string, since it is made up of a sequence of characters "strung" together. That is all there is to a definition of a string datum -- a string of printing characters that are considered as a single identifiable quantity. Other examples of strings are:

```
MY NAME IS
ABCDEFGHIJKLMNOPQRSTUVWXYZ
$HSL JMVM %$0/MM                Note that space is considered
XXXXXXXPPPPPQQQQQKKKKK          a string character.
10774992784993774552.HH GGY08   Note that printing
23.78                           numerals may be "strung"
0.27                            together to form a
1                               STRING datum.
5678.9953
```

The maximum length of any string is 73 characters.

Of course, the concept of string data would be of little use if there were no way to store and reference the particular string value of interest. Just as MUMPS allows the programmer to define a variable name that has a numeric value, it also allows him to define a string variable -- which naturally enough has a string value.

In order to define a string variable having a particular string value, you may use the SET command in much the same way that you used it to SET a numeric valued variable. For strings, however, you must notify MUMPS that the variable is to be assigned a string value rather than a numeric value. This can be accomplished by typing SET, followed by a space, followed by the name (which may be subscripted or unsubscripted) of the variable to be assigned a string value, followed by an equal sign (=), followed by a quotation mark ("), followed by the actual string to be assigned to the variable, followed by a closing quotation mark. For example:

```
>SET NME="JOHN DOE",AGE=22,STR="22"

>TYPE NME," ",STR,!,AGE
JOHN DOE 22
22
>
```

In this example, a single SET command has three arguments to consider. First, it SETs the variable NME to the string value "JOHN DOE". Secondly, it SETs the variable AGE to the numeric value 22. Note that there are no quotation marks in the AGE=22 portion of the statement. MUMPS assumes that the user means to assign the numeric value twenty-two to the variable named AGE. When MUMPS encounters the argument STR="22", it interprets it to mean SET the variable called STR equal to the string value formed by the concatenation of two numeral "2"'s. To understand the difference between the numeric value 22 and the string value "22" consider the two expressions below:

```
AGE+10*(I-1)/5.78

1+3*2.5+STR
```

The first expression can be evaluated and the result of the calculation could be used to SET some variable or type out the numeric result. The second expression, where STR is a defined STRING variable, is meaningless. One cannot combine a number value and a string value. When evaluating the second expression MUMPS would output the error message MIXED (both string and numeric values in the same expression). However, the members of an array need not be of the same data type. For example:

```
SET ID(1)="JOHN DOE",ID(2)=22,ID(3)="22"
```

is perfectly legitimate.

3.2 STRING CONCATENATION (.)

String values may be combined (concatenated) by use of the concatenation operator which is a period (.). For example:

```
>SET A="NME= ",B="FRED "

>SET A=A.B

>TYPE A."JONES"
NME= FRED JONES
>
```

3.3 STRING INPUT -- THE READ COMMAND

String data can be ENTERed and stored in the MUMPS memory with the READ command; READ is analogous to the ASK command used to input numeric data. In both cases, MUMPS waits for the user to ENTER a value, which is then stored as a variable. With the READ command, however, whatever data the user enters is assumed to be a string datum. The syntax for using the READ command is the same as for ASK. Like the ASK command, READ can force output of quoted text literals separated from the variable names by delimiting commas. For example, if you wanted to request the user's name, age, and address, you might use the following program:

```
9.10 READ !,"WHAT IS YOUR NAME PLEASE? ",NAM
9.12 ASK !,"AND YOUR AGE? ",AGE,!
9.40 READ "AND WHERE DO YOU LIVE? ",!,ADD

>DO 9

WHAT IS YOUR NAME PLEASE? JOHN DOE
AND YOUR AGE? 28
AND WHERE DO YOU LIVE?
1204 SOUTH BAY STATE ROAD, BOSTON, MASS.
>
```

When MUMPS encounters the READ statement above, it outputs the special format characters and the quoted text-literals, and then waits for the user to ENTER a character string as the answer to the question. When the user presses the ENTER key, MUMPS assigns the string value to the variable.

3.4 STRING COMPARISONS (=, [,],)

String values may be tested against other string values in much the same way that numeric values may be tested. The examples below assume an array named ID in core containing 20 names (held as strings) and 20 numbers (held as numerics). ID(1) contains the first string, ID(2) the first number, and so on up to ID(39), which contains the last string and ID(40) which contains the last number. String equality can be tested by using the equals operator (=). For example, the program:

```
4.10 READ "NAME=",NAM,!
4.20 FOR I=1:2:39 IF ID(I)=NAM TYPE ID(I+1),!
4.30 TYPE "END OF SEARCH"
```

will search the table and print out the number associated with the name that was input, providing that the name is found.

```
>DO 4
NAME=JACK JONES
103
END OF SEARCH
>DO 4
NAME=HARRY TRUMAN
END OF SEARCH
>
```

To make this program more useful, the contains operator ([) might be used; this tests whether the string value on the right side is contained in the string value on the left side. For example, assuming the program

```
5.10 READ "NAME=",NAM,!
5.20 IF NAM="" Q
5.30 FOR I=1:2:39 IF ID(I)[NAM T ID(I)," ",ID(I+1),!
5.40 TYPE "END OF SEARCH",! GOTO 5.10
```

is stored, then the following might occur

```
>DO 5
NAME=JACK JONES
JACK JONES 103
END OF SEARCH
NAME=JACK
JACK JONES 103
ALFRED JACKSON 104
JACKO SMITH 105
END OF SEARCH
NAME=ROG
END OF SEARCH
NAME=E
JACK JONES 103
ALFRED JACKSON 104
BILL RODGERS 101
END OF SEARCH
NAME=
```

The user simply strikes ENTER to return to direct mode; this inputs the <u>null</u> character represented in 5.20 by "" (that is, <u>nothing</u> enclosed in quotes).

Note that "JACK"["JACK" would match, and that "JOACK"["JACK" would not.

Suppose we wished to output all the names starting with G or H. The <u>follows</u> operator (]) can be used; this tests whether the string value on the left "follows" that on the right (i.e., whether the value on the left would appear after the value on the right if both were in a dictionary). To extend this to the full set of characters that MUMPS can store in a string, the following collating sequence is used:

```
null                                  ⎫
A through Z                           ⎪
[ ] ↑ ←                               ⎪  reading from left to
space                                 ⎬  right and from top to
! " # $ % & , ( ) * + ' - . /         ⎪  bottom
Ø through 9                           ⎪
: ; < = > ?                           ⎭
```

Thus "#" follows "A", "ABC" follows "AB" (the "C" is compared with null), etc. By convention, " FRED" does <u>not</u> follow "FRED".
To return to the example, the program

```
6.1Ø FOR I=1:2:39 DO 7
6.2Ø TYPE "END OF SEARCH",!

7.25 IF ID(I)]"I" Q
7.3Ø I ID(I)]"G" TYPE ID(I)," ",ID(I+1),!
```

will achieve the desired result.

```
>DO 6
GEORGE SMITH 124
GERALD JACKSON 144
HARRY THORT 165
END OF SEARCH

>
```

<u>Inequality</u> Conditions can be tested by use of the Boolean NOT operator ('). For example:

```
1.10 READ "CODE = ",C
1.15 IF 'C="272B" TYPE !,"ILLEGAL",! GOTO 1.10
1.20 TYPE !,"BEGIN"

>DO 1
CODF = 34
ILLEGAL
CODE = ROGER
ILLEGAL
CODE = 272B
BEGIN
>
```

In STEP 1.15, "ILLEGAL" is typed only if the contents of C equals something other than 272B (i.e., '(C=272B)).

The "not" operator can be used with the contains ([) and follows (]) operators as well.

3.5 PATTERN VERIFICATION (:)

Very often, in response to a stored READ command, the user will ENTER a string datum that is different from what the MUMPS program may be expecting. If the need for exact specification is real, the program may examine the value of a variable and make some decision based on the type of characters that the user entered. An example for this type of pattern recognition is the entry of a date in a program that is assigning hospital beds to incoming patients. For reasons of uniformity, the programmer may wish to insist that the user enter the date in the following pattern: two numbers for the month, a slash, two numbers for the day, a slash, and two numbers for the year; for example, 10/07/46 or 12/08/68. The pattern verification operator (:) can be used for this purpose.

The following program will READ the date and check it to make sure that the string value ENTERed corresponds to the pattern required:

```
1.33 READ !,"DATE OF ADM: ",DAT,!
1.40 IF DAT:2N"/"2N"/"2N QUIT
1.50 TYPE "NO GOOD, PLEASE FIX " GOTO 1.33
```

In Step 1.33, the string value of DAT is entered by the user. Step 1.40 checks the value of DAT to see if it is of the pattern 2N (two numerals), followed by a slash mark (the quoted text-literal "/"), followed by 2N (another two numerals), etc. If the value entered for DAT fails to meet the pattern specifications, control passes to Step 1.50 and a message is printed out.

```
>DO 1

DATE OF ADM: JAN 1 7Ø
NO GOOD, PLEASE FIX
DATE OF ADM: 1 JAN 7Ø
NO GOOD,PLEASE FIX
DATE OF ADM: Ø1/Ø1/7Ø

>
```

In actual fact, specific patterns such as that described above are rarely used. More often, it is desired to determine if the user has entered a specific type of string character. The pattern codes recognized by MUMPS are:

A	Passes alphabetic characters only (A-Z). Does not pass numbers or punctuation.
N	Passes numerals (0-9) only.
M	Mixed. Passes combinations of alphabetics and numerals.
P	Passes only punctuation marks. Includes all printing characters (including a space) except the numbers and the alphabetic characters.
Q	Passes combination of alphabetic and punctuation.
Z	Passes combination of numbers and punctuation.
U	Universal. Passes any character.

NOTE

A quoted text literal passes only the characters enclosed in the quotes (specifies an exact match).

In the example above, the pattern specification follows the single colon and contains numbers as well as the numeric pattern code (N for numerals). If the programmer does not care exactly how many numerals are entered, but merely wishes that only numerals will be passed, he may use the indefinite pattern match N, which is simply the code letter without any numbers in front of it. For example, if the programmer wishes to test whether or not the variable DAB is composed of an indeterminate number of alphabetic characters followed by precisely four numbers and one punctuation mark (and nothing else), the pattern A4N1P will serve the purpose. Notice that any number of alphabetics (even zero) will satisfy the first code letter. Some additional examples may be found below. Note that any given string can be passed by several different patterns. The more specific the programmer is in setting up his pattern match, the finer a filter will be implemented.

String	Some of the Patterns that will pass it
1234DD	NA, 4NA, 4N2A, "1234"A, "1"3N1A"D", U
JOHN DOE	U, APA, A" "A, A1P3A

Note that in using the pattern verification operator (also called the syntax analyzer) the string value must be on the left of the operator, and the pattern on the right. For example:

```
IF STR:N DO 7

IF QTY:2A1P3N DO 7

IF ABC:2N&MON:3A1P5N DO 7
```

In the first two examples, IF the variable name before the colon has the pattern coded after the colon, Part 7 will be "done". If the pattern does not match the value of the variable, Part 7 will not be done. In the third example, there are two pattern matches, connected by the Boolean operator & (AND). Both patterns must be satisfied in order to execute Part 7. Note that spaces may not explicitly appear in the pattern, except in a quoted text literal.

Strings can be tested for a "no match" condition by use of the Boolean NOT operator ('):

```
2.10 READ !,"LAST NAME, FIRST NAME ",N
2.15 IF 'N:A", "A GOTO 2.10
2.20 READ !," ID ",ID
```

This format is more convenient if you want to continue within a single part whether or not the "match" condition occurs.

MUMPS functions, which are identified by the $ prefix, are not described until Chapter 5. However, the $VALUE function is introduced here to show its use with string comparisons. The $VALUE function converts a string value to a numeric value.

Thus, $VALUE("33") will have the numeric value 33. If the string value does not represent a number, the value zero is returned. Both $VALUE("ØØ" and $VALUE("FRED") will have the numeric value Ø. When MUMPS executes a function it only examines the first character of the name after the $, and thus function names may be abbreviated in much the same way as commands. For example:

```
3.17 S A=$V(AGE)
```

The following example operates as an enquiry program for an array of names and numbers similar to that used in previous examples. If the user inputs a number, each name associated with that number is output (but in alphabetical order); if the user inputs a string containing a space, the number associated with first instance of that name is found. If a string not containing a space is input, every name containing that string is output, along with its associated number. Such a program might allow enquiries to an internal telephone directory; as it is coded for this example, it does not represent the best way of implementing such an application, but is given solely to illustrate the concepts discussed in this section. The example is coded in short form to give you an idea of how terse the MUMPS language can be. Note that the actual strings are not sorted, but rather only the pointers to them in array L.

```
1.1Ø R !,"QUERY :",S I S="" Q
1.2Ø I S:N D 2 G 1.1Ø
1.3Ø I S:A" "A DO 3 G 1.1Ø
1.4Ø S J=Ø F I=1:2:39 I ID(I)[S T !,ID(I)," ",ID(I+1) S J=J+1
1.5Ø I J=Ø T !,"NONE"
1.6Ø G 1.1Ø

2.25 S N=$V(S),J=1 F I=2:2:4Ø I ID(I)=N S L(J)=I-1,J=J+1
2.31 I J=1 T !,"NONE" Q
2.32 I J=2 G 2.5Ø
2.39 S T=Ø
2.4Ø F I=1:1:J-2 I ID(L(I))]ID(L(I+1)) S T=1 D 2.61
2.42 I T=1 G 2.39
2.5Ø F I=1:1:J-1 T !,ID(L(I))
2.51 Q
2.61 S K=L(I),L(I)=L(I+1),L(I+1)=K

3.5Ø F I=1:2:39 I ID(I)=S T !,ID(I+1) Q
3.69 I I=41 T !,"NOT FOUND"
```

```
←DO 1

QUERY :MIKE
NONE
QUERY :1004
DAN B.
DON U.
QUERY :1010
NANCY K.
WILLY W.
QUERY :DON
DON U. 1004
DON W. 1038
QUERY :BILL
BILL S. 1026
QUERY :JOHN
JOHN Q. 1030
JOHN M. 1042
QUERY :1060
NONE
QUERY :
←
```

CHAPTER 4

MORE ABOUT MUMPS PROGRAMMING

In previous chapters of this manual, little or no prior knowledge or experience with MUMPS was assumed. This chapter, however, as well as Chapters 5 and 6, assumes some familiarity with the MUMPS language. Basically, the purpose of this chapter is to introduce to the reader several additional commands and capabilities, and to tie up any loose ends created in the previous chapters.

4.1 MUMPS PROGRAM LIBRARIES -- CALL AND OVERLAY COMMANDS

The amount of core space allocated to any user is limited in size, and each program must fit into this space. However, a program may cause execution of other filed programs by use of the CALL and OVERLAY commands. When a program is brought into core by a CALL or OVERLAY command, it replaces the invoking program and MUMPS begins executing it at the first non-zero part. A program accessed in such a manner is treated like any other program by MUMPS, and it may CALL or OVERLAY still more programs. The effective size of the user's program is thus extended indefinitely.

CALL is treated in the same manner as the DO command, except that it takes program names as arguments instead of part numbers. When the program that was CALLed is finished executing, the original CALLing program is read back into core and re-entered at the point immediately following the invoking CALL command. However, the calling program itself must be filed, or MUMPS will be unable to find and return to it. (For details, see Chapter 7.)

The second command, OVERLAY, is used like the GOTO command previously discussed. Unlike CALL, the OVERLAY command cannot take more than one program name as an argument, since MUMPS does not "remember" the name of the OVERLAYed program, but simply reads in the OVERLAYing program and begins execution of it at the first non-zero part.

When programs are CALLed or OVERLAYed local data is available and remains unchanged; execution begins at the first non-zero part. When a program is brought in by OVERLAY in direct mode, control remains with the user in direct mode. Since each use of the OVERLAY command takes an average of one Disk access time, and since each CALL command takes two Disk accesses (one for the CALLed program, and one to return to the CALLing program), care should be exercised in the planning of program segments.

4.2 INPUT/OUTPUT DEVICES

MUMPS timesharing allows multiple users to have access to the same central processor via separate remote terminals. It also allows one user to have access to many terminals from one program. In addition to terminals, MUMPS systems also include ancillary Input/Output devices such as the high-speed paper tape reader and punch, DECtape transports, etc. Each of these I/O devices is assigned an identification number. To use one of these devices, you must perform the following:

First, you must establish "ownership" of the device since, in a timesharing environment, many programs may be competing for a single device. Ownership is established through use of the LOCK command. Before sending output to a device, you must LOCK to it. In direct mode this is done by entering:

```
LOCK DEVICE
```

where DEVICE is a numeric expression that represents the number of the device in question. Note that in direct mode the LOCK command must be the last command on a line. If the LOCK procedure is successful, that is, if no other user has already LOCKed to the device specified, MUMPS responds with a Carriage-Return and Line Feed and prints the usual right caret (>). If the device has already been LOCKed to, MUMPS responds by typing IOLOK, which indicates that the device is "owned" by some other program. If there is no such device, a MAXIM error printout occurs.

MUMPS follows a different procedure when the LOCK command is a stored command (i.e., in a Step). If the LOCK is successful, MUMPS <u>ignores</u> the rest of the line after the LOCK command and proceeds to the next Step in the program. If the LOCK procedure is unsuccessful, MUMPS continues on the same line and proceeds to execute any further commands. The usefulness of this procedure may best be illustrated by examples:

```
1.20 LOCK 12 TYPE "DEVICE 12 IS BEING USED" QUIT
2.39 LOCK 6 TYPE "6 IS IN USE, TRY ANOTHER" GOTO 5
4.56 LOCK 6 GOTO 4.56
```

In the first example, if the LOCK is unsuccessful, MUMPS continues on the same line, outputs the quoted message, and then QUITs the part it was executing. In this case, the programmer has decided that if he cannot have device number 12, then he does not want any device (so he QUITs). In the second example, the same thing happens, except that in this case the programmer decided to try another, presumably in Part 5. The technique illustrated by the third example is much more common; if the LOCK is unsuccessful, the program continues trying until it is successful. (Note that a program may be required to wait for an indeterminate amount of time using this technique.) In each of the three examples, if the LOCK is successful, control passes to the next Step. A program may own more than one I/O device at a time, but a LOCK command takes only one argument.

After the program has established ownership of a device, it must then inform MUMPS that it wishes to use the device for actual I/O. A program may not communicate simultaneously with more than one input/output device. The device identification number is stored in a local variable called %I. This variable may be treated as any other numeric variable in local storage. It may be referenced in a number-valued expression, and it may be SET. In order to change the device being used for I/O, one need only SET the value of %I to the desired ID number. Each I/O command references the device whose ID number equals the current value of %I.

The PRINCIPAL I/O DEVICE is defined as the device (usually a Teletype-like terminal) that originally calls up the program. When %I is equal to zero, MUMPS uses the principal device, whatever it happens to be. It is the custom when using the terminal to SET %I equal to zero. When a user signs on from the terminal, %I contains the actual device number divided by one hundred (this allows the program to identify the terminal being used). When MUMPS detects an error, the value of %I is SET equal to zero so that error messages are output to the principal I/O device.

When an I/O device is no longer required by a program, it can be released for use by other programs by means of the UNLOCK command. UNLOCK may take a list of arguments and UNLOCKs are always successful; MUMPS continues on the same line of code after executing the UNLOCK. When a program is halted, all devices "owned" by that program are UNLOCKed automatically. LOCKing and UNLOCKing the principal I/O device are ignored.

4.3 SECONDARY STORAGE

Obviously, the limited amount of fast core memory available can not be allocated for applications requiring vast amounts of storage. Consequently, core is partitioned among the users, and only a limited amount of fast local storage space is available per user. MUMPS provides three types of secondary storage systems: Global arrays stored on Disk, linear storage tracts stored on DECtape, and paper tape.

The structure and use of Global arrays is discussed in Chapter 6. For the time being, let us turn our attention to the use of DECtape devices.

4.3.1 DECtape I/O

Multiple DECtape transports are available for use under program control. Access to a DECtape is obtained by setting the Input/Output device variable (%I) equal to the appropriate device identification number. Subsequent READ, WRITE, ASK, and TYPE commands operate on the DECtape unit specified by the value of the %I variable. Since the DECtape is organized as a linear storage device, one additional control variable is needed. The address variable %ADDRESS (%A), which can be set by the user, points to the next word position that will be accessed by any of the commands that require I/O. The value of %A is in terms of DECtape words divided by one hundred so that %A=24.05 means that the next word position to be referenced is number two thousand four hundred and five. Each DECtape word holds three MUMPS characters. As MUMPS accesses words, the value of %A is updated so that it always points to the next available location. The range of %A is 0.00 to 1310.71. The availability of the address variable permits extensive use of address arithmetic in applications needing large amounts of data for long term storage.

All output to DECtape must be followed by an EOM, symbolized by an exclamation point (!) or a page feed (#). For example:

```
>TYPE X,!,Y,!,Z,!
```

If a TYPE command is not followed with an EOM, the next attempt to TYPE onto DECtape will cause an error (even though that next statement is correct).

4.3.2 Paper Tape I/O

The paper tape station is programmed in much the same way as terminals. After LOCKing to the device and setting up the %I variable, TYPE, PRINT, READ, WRITE, and ASK commands are all legal. Paper tape provides a convenient off-line storage medium for programs during early stages of development.

4.3.3 Saving Programs Using READ and WRITE Commands

Programs may be saved and retrieved on either paper tape or DECtape mediums by means of READ and WRITE commands without arguments. It is only necessary to LOCK to the device and set the %I variable appropriately. For example, let us assume that in a particular MUMPS configuration the paper tape station is assigned as device number 12. The following command sequence, then, will "dump" any locally held MUMPS program onto paper tape.

```
>LOCK 12
>SET %I=12 W
```

The same program may be retrieved at a later time and restored within the user's partition by the following sequence.

```
>LØCK 12
>SET %I=12 R
```

Programs can be saved on DECtape by similar setup commands. However, the %A variable may need to be set to a starting value, and it is advisable to add an "end-of-program" marker. For example, assuming the DECtape is device No. 14, the commands:

```
>LOCK 14
>SET %I=14,%A=24.Ø5
>W  T "*",!
```

cause the local program to be written on DECtape unit 1 starting at word 2405. After the "write all" (WRITE followed by double space), an asterisk is added as an "end-of-program" marker.

The program may be retrieved and restored in the user's partition by the following sequence:

```
>LOCK 14
>SET %I=14,%A=24.Ø5 R
```

When the program is read back, the asterisk creates an error because it is not a step number. The error stops the tape at the point where one program ends and the next begins.

4.4 SPECIAL (SYSTEM) VARIABLES

A number of special system variables are defined within MUMPS to control the flow of information and to provide system information to individual users. Two of these variables (%A and %I) have already been discussed; all of the special variables are listed and defined in tables 4-1 and 4-2. Table 4-1 defines variables that may either be referenced or SET by the user. Table 4-2 defines special variables that may only be referenced by the user. You will note that each variable shown in Table 4-1 begins with a "percent" character (%), and each variable in Table 4-2 begins with a "dollar sign" ($) character.

Table 4-1. Special Variables that may be Referenced or SET

VARIABLE	ABBREVIATION	DEFINITION	
%ADDRESS	%A	Address of next word to be accessed on DECtape. (Automatically updated after a DECtape access.)	
%ERROR	%E	Controls action of independent program when errors or interrupts occur. (Initially -.01)	
		%E=Ø	BREAK is ignored.
		%E=-.Ø1	BREAK terminates the program.
		%E=-.Ø3	Set %E to Ø on a BREAK. (Use for implementation of user defined operations.)
%IODEVICE	%I	Contains the number of the I/O device to be used by MUMPS. When the job is first started, %I contains the device number divided by 100.	
%FORMAT	%F	Contains the number of characters from the left margin that define the location of the right margin. (Initially set to 72.) CRLF occurs on first space following the %F-th character.	
%MARGIN	%M	Controls where the first character on a line will occur; can be used for tabbing. %M should be less than %F for proper operation.(Initially Ø.)	
%PAGE	%P	Controls where a carriage return and a form-feed control character is output in lieu of a CRLF. %P=Ø disables this feature. (Initially Ø.) (Form-feed is not available on all types of terminal.)	
%KEY	%K	Contains protection key for globals or programs. (Initially Ø.) Refer to Chapter 7 for more detailed information on key protection.	

Table 4-2. Special Variables that may be Referenced Only

VARIABLE	ABBREVIATION	DEFINITION
$LOCATION	$L	Equals the number of the Step that is being executed at any time.
$STORAGE	$S	Equals the number of free characters of storage left in the partition.
$TIME	$T	$TIME contains the number of seconds since midnight, divided by 100.
$DATE	$D	$DATE contains a six character string with year, month, and day; "yymmdd". This form allows convenient collating operations.
$X	$X	X-coordinate. Equals the number of characters typed since the last carriage-return line-feed on the principal output device only.
$Y	$Y	Y-coordinate. Equals the number of carriage-return line-feeds since the last PAGE feed on the principal output device only.

4.5 SPECIAL COMMANDS AND COMMAND FORMS

In addition to the commands described in preceding sections of this manual, MUMPS provides several other special commands and command forms that are described in the following paragraphs.

4.5.1 The HANG and HALT Commands

The HANG command provides for a program to "HANG" for a specified number of seconds during which time no attention is given to the program calling for a HANG. This facility is especially useful in applications where the programmer periodically wants to check the status of a variable and take action when the variable has changed; in the event that it has not, the program goes into "limbo" for another period of time. The number of seconds to HANG is specified as an argument to the HANG command. Consider the following examples:

HANG 0.04	The next command is not executed until four seconds have elapsed.
HANG 1.20	The next command is not executed until 120 seconds have elapsed.
HANG 0	Give up the rest of your current processor time slice.
HALT	Terminates the user's session at the terminal.

The reader will note the similarity between HANG and HALT; that is, both commands begin with the letter H. If a command begins with an H and has no argument after it, MUMPS interprets it as HALT. If there is an argument following the command, MUMPS interprets it as a HANG command and takes the appropriate action.

4.5.2 More About the QUIT Command

To better understand the QUIT command, it may be helpful to think of a program's execution as occurring at different levels. Each time a DO command, a CALL command, or a FOR modifier is encountered the level of the program is raised by one. The normal termination of the above commands and FOR modifier decreases the current level by one. When QUIT is executed, the current level is also decreased by one and the invoking command or FOR modifier is terminated.

The QUIT command is used if a program is in the range of a DO or CALL command and it completes its computation; control then returns to the code immediately following the invoking command. If the invoking command was a DO from direct mode, control is returned to direct mode. If the program is currently in a FOR loop, the repetitive loop ends when a QUIT command is executed, and the program continues from the QUIT statement. For example:

```
1.01 F I=1:1:10 R !,A(I) I A(I)="DONE" Q T !!,"THANKS"

>D 1.01

10
20
30
40
50
DONE

THANKS
>
```

The array A(I) is READ until A(I) is equal to "DONE", at which point the FOR is terminated, and the message "THANKS" is typed.

4.5.3 More About the DO Command

There are two additional points that should be made concerning the use of the DO command that were not covered in Chapter 2. First, for some editing applications, the user may wish to create a new program Step under program control rather than by entering the new Step via direct mode. This may be accomplished by use of the DO command if the argument is a STRING of the proper format. The use of this facility may best be illustrated by examples:

Statement	Effect
DO "1.20 TYPE X+3"	A Step 1.20 is created containing the command TYPE X+3.
DO 3.1Ø,$C(I)." ".STP,3.2Ø	Step 3.1Ø is "done", a Step created whose contents are the value of the string STP, and then Step 3.2Ø is "done".
	The Step number of STP is the value of the number variable I. Note the mandatory space between the Step number $C(I) and the contents STP.
DO $C(I)." ".STP,I	As above, Step I is created. But in this case, the DO command has a second argument, I. Therefore, this command will create the Step and then immediately execute it.

Secondly, it was stated in Chapter 2 that if the argument of a DO command was a Step number, then MUMPS would execute that Step and then return to the next argument, if any, in the DO command. This is strictly true only if the Step that was "done" does not have any GOTO command in it. If the invoked Step does have a GOTO in it, MUMPS will honor the GOTO and will not return to the invoking DO command until the end of a Part or a QUIT command is encountered, after which it returns as usual. In effect, this gives the programmer the ability to DO a Part starting with a Step that is not necessarily the first Step in that Part.

4.5.4 UNTIL and WHILE Terminators in the FOR Clause

The FOR statement has been described as having two different formats (Range and List) that may be intermixed. For example: 1:1:200 (Range), 1,4,2,8,45 (List), 1:1:4,8,12 (intermixed), and 1,5,2:4:12,3,6,1:1:5 (intermixed).

In the Range format, the limit may be replaced by a WHILE or UNTIL clause. For example:

For K=1:Ø UNTIL X=2...	MUMPS will do everything in the range of the FOR statement UNTIL the Boolean condition (X=2) is TRUE. It then stops and goes on to the next Step.

FOR K=1:1 WHILE X=2...	In this case, MUMPS will do everything in the FOR statement WHILE the Boolean condition (X=2) is TRUE. In other words, when the Boolean condition <u>fails to be true</u>, the FOR statement will be stopped and control passes to the next line.

4.5.5 The PRINT Command

Although MUMPS has been created with a great deal of device independence, it is very often desirable to take advantage of special features of certain I/O devices. The PRINT command accepts numeric arguments, the low-order seven bits of which are taken as an ASCII character. This character is transmitted without any conversion to the device specified by %I. Using this command, it is possible for the programmer to take advantage of the control functions for a particular device, such as the bell (PRINT .07), a carriage return without a line feed (PRINT .13), or a line feed without a carriage return (PRINT .10). Multiple codes can be specified by separating them with commas. For example,

```
>PRINT .10,.10,.10,.07
```

results in three line feeds (without a carriage return) and ringing the bell.

4.5.6 XCOM Command

This command permits a MUMPS program to access specific disk blocks on either DECdisk or Disk Pack. The command is provided primarily for use by system programming or management personnel and is key protected for system security. Complete information on the use of t this command is provided in Chapter 7 of <u>MUMPS Operator's Guide</u> (DEC-15-MMUPA-A-D).

CHAPTER 5

USING FUNCTIONS IN MUMPS

MUMPS contains a number of useful functions. By convention, MUMPS function names are always preceded by a "dollar sign" ($) and, like commands and special variables, only the first letter of the actual name is required. Expressions that are to be evaluated as "arguments" by the function are specified in parentheses and separated by commas as part of the function call. The form for using functions is best illustrated by an example:

```
1.10 FOR I=1:1:10 SET ARY(I)=(2*I)*(2*I)
1.20 FOR I=1:1:10 TYPE " ",$ROOT(ARY(I))

>DO 1
 2 4 6 8 10 12 14 16 18 20
>
```

MUMPS provides for two basic types of functions that are classified according to the type of value that is returned. The Numeric Functions are those that return numeric values. Similarly, functions that return string values are termed String Functions. Numeric functions may be used anywhere numeric values are legal. String functions may also be used where string values are legal. However, except for $BOOLEAN, they may *not* be nested within other functions. Table 5-1 (at the end of this chapter) shows functions which may be legally nested.

In the function descriptions that follow, SVE refers to a string valued expression and NVE refers to a numeric valued expression. In cases where a function has more than one argument of the same type, the arguments are numbered according to position.

5.1 NUMERIC FUNCTIONS

Numeric functions included in MUMPS are:

$ROOT	$DEFINE
$LENGTH	$NEXT
$VALUE	$HIGH
$INTEGER	$OBTAIN
$FIND	$QUERY
$BOOLEAN	$Z
	$KORE

Each of these functions is described in paragraphs that follow.

5.1.1 $ROOT(NVE)

This function first forms the value of the numeric valued expression specified by NVE. NVE may be a number, a numeric variable, or an expression such as AGE+3/YRS, provided that the variables AGE and YRS both have numeric values. In any case, if the NVE results in a legally defined MUMPS number, $ROOT(NVE) returns the value of the square root of NVE. If NVE is a negative valued expression, MUMPS types the error message MINUS. $ROOT may be used in the same manner as any other numeric expression, for example, to compute and store the value of the hypotenuse of a right triangle of sides A and B one might use $ROOT in the following way:

```
SET C=$ROOT(A*A+B*B)
```

To determine if the hypotenuse is an even multiple of 5, one might use the following:

```
1.10 SET D=$R(A*A+B*B)/5
1.50 IF D/100*100=D TYPE "MULTIPLE" QUIT
1.60 TYPE "NOT MULTIPLE OF 5"
```

5.1.2 $LENGTH(SVE)

The $LENGTH function ($L) returns the number of characters contained in the string valued expression SVE. For example:

$L("HELLO")	Returns the numeric value 5.
$L(NAM)	Where the argument NAM has the string value "JOHN DOE", the numeric value returned is 8.

5.1.3 $VALUE(SVE)

The $VALUE function ($V) evaluates the string argument SVE and attempts to convert it to a number equal to the value represented by the digits in the string. If the leftmost characters of the string represent a valid number, $V returns that number; otherwise, $V returns Ø. The user should check for SVE="Ø" before using $V to avoid ambiguity. The following are all valid SVE's for the $V function:

$V("1.1Ø")	Returned as the numeric 1.1Ø.
$V("1+3/4")	Returned as the numeric 1.75.
$V(AGE)	Where AGE = "59", $V returns the numeric 59
$V("$T/6Ø,FRED")	Number of minutes since midnight/100

The following examples cause $VALUE to return the numeric value Ø.

$V("JOHN DOE")	This string does not represent a number.
$V(AGE)	Where AGE does not contain a string that can be evaluated as a number (e.g., "FIFTY-NINE YEARS.")

The $V function is useful after checking the syntax of a response to a query. In the following example, assume that a two-digit number is required:

```
1.1Ø READ "NUMBER PLEASE ",STR!
1.12 IF STR:2N TYPE "THANK YOU" GOTO 1.16
1.14 TYPE "USE ONLY TWO DIGITS PLEASE",! GOTO 1.1Ø
1.16 SET NUM=$V(STR)

>DO 1
NUMBER PLEASE 365
USE ONLY TWO DIGITS PLEASE
NUMBER PLEASE 35
THANK YOU
>
```

The reader will note that in applications programs it is safer to use a READ command followed by appropriate syntax checking than to use an ASK command. This technique allows programmed recovery from input errors.

5.1.4 $INTEGER(NVE)

This function returns the integer portion of the numeric valued expression NVE (i.e., the fractional part, if any, is not returned). For example:

$I(30.26)	Returns the numeric value 30.
$I(AGE/5)	If AGE contains the numeric value 59, the integer value returned is 11.

5.1.5 $FIND(SVE1,SVE2,NVE)

The $FIND function ($F) searches for the occurrence of the string SVE2 within SVE1. NVE is an optional argument and, if present, the search through SVE1 begins with the NVEth character; otherwise, the search begins with the first character of SVE1. If $F "finds" SVE2 within SVE1, it returns a number that represents the character position following the last character of SVE2. If $F does not find SVE2 within SVE1, it returns a zero. For example, where

STR="ABCDEFGHIJKLMNOP"

the following $F functions will return the values indicated.

$F(STR,"A",1)	returns 2.
$F(STR,"A",3)	returns Ø; A does not occur after the third character in STR.
$F(STR,"EFG",1)	returns 8.
$F(STR,"ACD",1)	returns Ø; the string ACD is not found.

5.1.6 $BOOLEAN(BOOL1,NVE1;BOOL2,NVE2;...;BOOLn,NVEn)

Very often it is desired to type one of two messages or to branch to different Steps depending on some condition in a program. This type of operation, where it is desired to do "A" if one thing is true, "B" if something else is true, or "C" if another condition is true, can be achieved with the $BOOLEAN function. The $B function tests multiple Boolean expressions and as soon as it finds one of the expressions true, returns a corresponding numeric value. For example, assuming X=1, Y=2, E="ABE", then

```
>T $B(X>Y,1;X=3,X+Y;E["A",X-Y;1,Ø)
-1
>
```

results in evaluation of the expression X-Y and typing of -1.

If the returned numeric value is a part or step number, it can be used as the argument to a GOTO, in which case a controlled branch is executed. It could also be used as the third argument of a $PIECE function where the string variable contains multiple messages and the $B controls which message the $PIECE would return. For example, if the variable X contains "HELLO, GOOD BYE", and the variable D controls the typing of either HELLO or GOOD BYE, depending on whether it contains a 2-digit number, then:

```
1.10 T $P(X,",",$B(D:2N,1;1,2))
```

would result in typing HELLO if D contains a 2-digit number, and GOOD BYE otherwise.

The $B function allows argument pairs to be separated by either commas or semicolons. The advantage in using semicolons is that the eye more easily can spot the separation of pairs than it can with commas. The function does not evaluate any of the numeric arguments except those within the NVE which it is returning. This allows the other NVE's to contain undefined variables. One requirement of the function is that there _must_ be at least one TRUE Boolean expression; therefore, it is suggested that the last Boolean argument contain an always TRUE value, such as the number 1.

5.1.7 $DEFINE (variable name)

The $DEFINE function ($D) can be used to check the data type of either local or global variables. The variable to be checked is simply named as an argument to the $D function. $D returns one of six number values as follows:

Ø	undefined	
1	string valued datum	
2	numeric valued datum	
3	(spare)	returned for global variables only (Ch. 6)
4	pointer to structure at lower level	
5	pointer and string valued datum	
6	pointer and numeric valued datum	

5.1.8 $NEXT(NVE)

$N returns the number of the first MUMPS Step following the Step specified by NVE. For example:

$NEXT(Ø)	returns the Step number of the first Step in the program.
$NEXT(Ø.99)	returns the Step number of the first Step in Part 1.

If there are no Steps after the value of NVE, $N returns zero (Ø).

5.1.9 <u>$HIGH(VAR(SUBSCRIPT))</u>

$HIGH returns the value of the lowest subscript numerically greater than the specified subscript of the given array. For example, if the array name ABC has the following defined values:

```
ABC(1)
ABC(1.5)
ABC(3.12)
```

then the following values are returned by $H:

```
$H(ABC(Ø))          returns 1.
$H(ABC(1.1))        returns 1.5.
$H(ABC(3.12))       returns Ø.
```

Notice that if there are no numerically greater subscripts, $H returns zero (Ø).

5.1.10 <u>The $OBTAIN and $QUERY Functions</u>

It is frequently useful to retrieve information from a global according to its physical arrangement rather than the logical arrangement. MUMPS does not physically sort a new entry at a level into ascending subscript order, but finds a "convenient" place within the file at that level. Therefore, while a file may logically consist of entries at subscripts 1, 3, 7, and 15, the physical layout may very well be 3, 1, 15, 7, in that order. As the file at that level gets larger, several disk blocks may have to be allocated for it, and it would be useful to know the physical order of the entries to minimize the number of disk accesses.

In many cases, it is necessary to consult every entry of a file at a certain level, in order to establish a successful "hit" (e.g., in the case where the first three letters of a surname are mapped to the first three levels, and the balance of the surname is stored, along with others having the same first three letters, in a 4th level sequential file). The $OBTAIN and $QUERY functions have been designed to aid the MUMPS programmer when he encounters this need.

5.1.10.1 <u>$OBTAIN (GLOBAL NAME ($SUB_1$,...,$SUB_n$))</u> -- $OBTAIN causes a search of the specified global array to be performed to determine if a level of subscripting exists below that specified in the argument (SUB_n). If the search is successful, the number of the (physically) first subscript or node is returned. If only an empty node exists (all data has been KILLed), a -1 is returned. If there is no lower level present, an UNDEF error results. For example, if there exists a global called "A" having the following physical structure:

```
↑A(3)          (pointer only)
↑A(3,7)        "String A"
↑A(3,2)        "String B"
↑A(3,4)        "String C"
↑A(3,1)        "String D"
```

The command:

```
SET J=$OBTAIN (↑A(3))
```

causes J to be set to the numeric value 7 and sets up the global for access at this level (i.e., naked syntax references can now be made at this level). If, in the above example, the command:

```
SET J=$OBTAIN (↑A(3,4))
```

is issued, an UNDEF will be caused since there are no lower levels defined in the structure. In practice, the user should utilize the $DEFINE function if there is any doubt as to the existence of a lower level.

5.1.10.2 <u>$QUERY (GLOBAL NAME ($SUB_1$,...,$SUB_n$))</u> -- $QUERY causes a search of the specified global array to be performed at the current level to obtain the next (physically sequential) subscript. If the search is successful, the number of the next subscript is returned; otherwise, a -1 is returned.

This function is typically used in conjunction with the $OBTAIN function to sequentially search a level completely in the physical order of subscript occurrence. The following example shows a typical application of both functions.

Assume the global name is ↑A, and that I, J, and K are subscript values for the first three levels. Further, assume that X contains the string to be searched for at the fourth level. D is a temporary variable for intermediate results.

```
19.10    SET D=$D(↑A(I,J,K))
19.20    IF D=4! D=5! D=6 DO 20 QUIT
19.30    TYPE "NOTHING TO SEARCH"

20.10    SET D=$OBTAIN(↑(K))
20.20    IF D=-1 TYPE "NOTHING FOUND" QUIT
20.30    IF X=(D) TYPE "FOUND IT" QUIT
20.40    SET D=$QUERY(↑(D)) GOTO 20.20
```

Comments:

1. Part 19 proves there is a structure to search and calls on part 20 to do the search.
2. Step 20.10 causes the global to advance to the fourth level as well as get the first physical subscript at that level.
3. Step 20.20 checks for end of file.
4. Steps 20.30 and 20.40 can use the naked access at the fourth level.

5.1.11 $Z(NVE)

The $Z function returns a pseudo random number of modulus NVE.

5.1.12 $KORE

The $KORE function is a feature of MUMPS which permits a program to access any specified core location. $KORE should only be used in conjunction with an assembly listing of the operating system and is provided primarily for use by system programming or management personnel. This command is key protected for system security. Complete information on the use of $KORE is provided in Chapter 7 of MUMPS Operator's Guide (DEC-15-MMUPA-A-D).

5.2 STRING FUNCTIONS

String Functions included in MUMPS are:

$CHARACTER
$EXTRACT
$PIECE
$STEP
$TEXT
$MONEY

> NOTE
>
> String functions cannot be nested together.

Each of these functions is described in the paragraphs that follow.

5.2.1 $CHARACTER(NVE)

The $CHARACTER function ($C) converts the number value NVE to a character string. For example:

$C(33)	yields the character string "33".
$C(X)	yields the character string "1Ø.92" where X=10.92.

The resulting string expressions can be evaluated by string comparison operators (:, =, [,]) for format control, etc.

5.2.2 $EXTRACT(SVE,NVE1,NVE2)

$E EXTRACTS all the characters from SVE that are between the NVE1st and NVE2nd character locations, inclusive. If NVE1 is greater than NVE2, $E returns a "null string". The null string can be represented in MUMPS programs by the notation "". If NVE2 is equal to NVE1 or if NVE2 is omitted, $E returns the NVE1st character only. If the length of the string is such that $E runs out of characters before satisfying both NVE1 and NVE2, then the function returns any characters between the NVE1st character and the end of the string. Only the integer parts of NVE1 and NVE2 are considered.

This function is frequently used in editing routines. By combined use of the $EXTRACT function with the concatenation operator (.), any segment of a string may be extracted and replaced by another. For example, assume that the string variable NAM="JOHN DOE" and we want to change it to last-name first, comma, first-name last. The following statements will do it:

```
SET LST=$EXTRACT(NAM,$FIND(NAM," ",1),$LENGTH(NAM))
SET FIR=$EXTRACT(NAM,1,$FIND(NAM," ",0)-2)
SET NAM=LAS.",".FIR
```

5.2.3 $PIECE(SVE1,SVE2,NVE1,NVE2)

Often, to save space, a programmer will pack several "pieces" of data in a single string, thus cutting down on overhead space and access time. The $P function examines the string variable SVE1, which is assumed to be divided into "fields" delimited by the first character of SVE2. $P returns the text in the fields specified by the two arguments NVE1 and NVE2. If NVE2 is equal to NVE1, or if NVE2 is omitted, $P returns only the NVE1st field without delimiters. If there is no NVE1 field, then a null string is returned. If $P runs out of fields before satisfying NVE2, it returns any characters between the SVE2 delimiter and the end of the string. For example, where

```
STR="34,6.Ø9,JOHN DOE,BOSTON,JUNE,22"
DEL=","
```

then the following would be true:

$P(STR,DEL,3)	returns "JOHN DOE"
$P(STR,DEL,2)	returns "6.Ø9"
$P(STR,DEL,3,4)	returns "JOHN DOE,BOSTON"
$P(STR,DEL,4,1Ø)	returns "BOSTON,JUNE,22"
$P(STR,DEL,8)	returns a null string, since there is no eighth field in SVE1
$P(STR," ",1)	returns "34,6.Ø9,JOHN"

The delimiter may be specified literally, as well as by a variable. If the delimiter within the argument is a punctuation mark, the same mark can be used as the delimiter in the $P statement without quotation marks:

```
        $P(STR,3)       returns "JOHN DOE"
```

5.2.4 <u>$STEP(NVE)</u>

The $STEP function ($S) returns the string value of the Step specified by NVE, <u>without</u> the STEP NUMBER. This function may be useful in editing applications where it is necessary to treat the contents of a Step as a string.

5.2.5 <u>$TEXT(NVE)</u>

The $TEXT function evaluates the numeric argument NVE to return three ASCII characters. $T is used primarily by MUMPS system programmers. For example, assume NBR is equal to 61Ø1Ø2$_8$. Then

```
                    $T(NBR)
```

returns the three-character string 1AB.

5.2.6 $\text{\$MONEY}(\text{SVE}_1 \left| \overbrace{\left\{\begin{array}{c}+\\-\\*\\/\end{array}\right\} \text{SVE}_2 \left\{\begin{array}{c}+\\-\\*\\/\end{array}\right\} \text{SVE}_3 \cdots}^{\longleftarrow\ \textit{optional}\ \longrightarrow} \right|)$

$MONEY extends the normal range of MUMPS numbers (±1310.71), to permit operations on numbers which contain up to nine significant digits and exponents in the range from 10^{46} to 10^{-30}.

The $M function allows the arithmetic operations +, -, *, /, where each of the operations has its normal arithmetic meaning. The $M function accepts only string valued expressions (SVE's), not actual numeric values. Hence, constants used in $M expressions must be enclosed in quotes and variables must have been previously set to string values. For example:

```
SET    X="1.5"
SET    Y="-2.3"  (Sign needed only if negative)
SET    Z="4"     (Decimal point may be omitted)
SET    SUM=$M (X+Y+Z)
```

The value of SUM will be the string "3.2".

```
SET  T=$M (SUM + "1.2")
```

Since 1.2 is a constant used in a $M expression it must be enclosed in quotes. Further, other MUMPS functions must not be nested within $M expressions. This means that the expression:

```
1.23 SET A=$M("123.4"+$C(72))
```

is illegal. However, the following will solve this problem:

```
1.23 SET B=$C(72)
1.33 SET A=$M("123.4"+B)
```

Unlike normal MUMPS arithmetic operations, a $M expression is evaluated STRICTLY from left to right, in the order in which the operations appear. Parentheses are NOT permitted. Thus, a $M expression containing multiplication and division may indeed yield a different result from that given by normal MUMPS arithmetic. For example:

```
SET A=6, B=9, C=3
SET X=A+B/C                   ;RESULT X=9

SET A="6", B="9", C="3"
SET X=$M (A+B/C)              ;RESULT X="5"
```

In the first example, because it is normal MUMPS arithmetic, B/C is evaluated prior to addition to A. Hence, B/C = 3, A+3 = 9. In the second example, because it is $M evaluation, A + B is evaluated first and then that result is divided by C. Hence A + B = "15", 15/C = "5", which is the STRICT left to right application of the arithmetic operations.

If the desired result was $X = A + \frac{B}{C}$, two solutions are possible:

```
SET    T = $M (B/C)
SET    X = $M (A+T)
```

or

```
SET    X = $M (B/C + A)
```

In many instances the second solution, which is obviously more desirable because it is shorter, will suffice. In complicated expressions it may be necessary to defined sub-expressions and then use these in a $M function involving only addition and subtraction.

In addition to nine digits and an optional sign and decimal point, string values to be used in $M functions may include an exponent. The exponent portion consists of an up arrow (↑), a sign, and one or two digits indicating a power of 10.

```
1.11   SET    X="15↑+5"
1.21   SET    Y="-1.275↑-12"
```

In the first example the result is 15 x 10^5 or 1500000. In the second example the result is -1.275 x 10^{-12} (or -0.000000000001275).

When it is necessary to input a value to be used in a $M function, it should be input via a READ command rather than an ASK, since the READ allows the input of string valued variables. When entering data via READ, the string may contain up to nine significant digits, an optional decimal point, and sign, and an exponent. When more than nine significant digits are entered all digits following the ninth are interpreted as an equivalent number of zeroes.

INPUT VALUE	VALUE USED FOR CALCULATION
"1234.56"	123.456
"36942305.009"	36942305
"123456789123↑-13"	.0000123456789
"600430009523760"	600430009000000
".00032405619342"	.000324056193

When values calculated by $M functions are to be printed, they are entered in TYPE statements just as any other numeric output. Up to nine significant digits are output with a sufficient number of leading or trailing zeroes to adjust for the power of ten required.

RESULTS OF CALCULATION	OUTPUT VALUE
1.23456	1.23456
1.5X10^{15}	1500000000000000
6.78X10^{-12}	.00000000000678

The following examples illustrate many of the features of the $M function.

Example A:

The following program will input a value N and compute N factorial. N factorial is defined as the product of all

integers between 1 and N. It is written symbolicially as N!
Hence 5! = 1 X 2 X 3 X 4 X 5 = 120.

```
1.1∅  READ !,"ENTER N.  PROGRAM WILL COMPUTE N! ",N,!
1.2∅  SET X="1",FAC="1"
1.25  SET NN=$V(N)
1.3∅  FOR I=1:1:NN SET FAC=$M(F*X),X=$M(X+"1")
1.35  TYPE N,"FACTORIAL=",FAC,!
1.4∅  GO TO 1.1∅

DO 1

ENTER N.  PROGRAM WILL COMPUTE N!  5
5 FACTORIAL=12∅

ENTER N.  PROGRAM WILL COMPUTE N!  1∅
1∅ FACTORIAL=36288∅∅

ENTER N.  PROGRAM WILL COMPUTE N!  15
15 FACTORIAL=13∅7674368∅∅∅
```

Example B:

The following program allows the user to enter a value R which is the radius of a circle. The program calculates the area of a circle of radius R from the equation $A=\pi R^2$.

```
1.1∅  SET PI="3.14159"
1.2∅  READ "ENTER R ",R,!
1.3∅  SET A=$M(PI*R*R)
1.4∅  TYPE "FOR A CIRCLE OF RADIUS ",R,"THE AREA IS ",A,!!!
1.5∅  GOTO 1.2∅
```

Example C:

The program is a compound interest problem. The input is a rate of interest as a percentage. The program determines the number of years it takes to double $100 at the given rate by compounding interest annually.

```
1.∅5  SET DBL="2∅∅"
1.1∅  READ !!!!,"ENTER INTEREST RATE ",R,!
1.2∅  SET C="1∅∅",Y=∅
1.3∅  TYPE !!,"CAPITAL INTEREST YEAR",!
1.4∅  SET INT=$M(C*R),C=$M(C+INT),Y=Y+1
1.5∅  TYPE C,"    ",INT,"    ",Y,!
1.6∅  SET X=$M(C-DBL) IF $E(X,1)="-" GOTO 1.4
1.7∅  GOTO 1.1
```

TABLE 5-1

LEGAL FUNCTIONS FOR NESTING[††]

PRIMARY FUNCTION		FUNCTION TO BE NESTED																	
		NUMERIC													STRING				
		$BOOLEAN	$DEFINE	$FIND	$HIGH	$INTEGER	$KORE	$LENGTH	$NEXT	$OBTAIN	$QUERY	$ROOT	$VALUE	$Z	$CHARACTER	$EXTRACT	$MONEY	$PIECE	$STEP	$TEXT
NUMERIC	$BOOLEAN	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	$DEFINE[†]	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$FIND	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$HIGH[†]	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$INTEGER	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$KORE	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$LENGTH	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
	$NEXT	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$OBTAIN[†]	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$QUERY[†]	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$ROOT	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$VALUE	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
	$Z	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
STRING	$CHAR-ACTER	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$EXTRACT	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$MONEY	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
	$PIECE	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨
	$STEP	X	▨	▨	X	▨	▨	▨	X	▨	▨	▨	X	▨	▨	▨	▨	▨	▨	▨
	$TEXT	X	X	X	X	X	X	X	X	X	X	X	X	X	▨	▨	▨	▨	▨	▨

[†]Nesting is permitted only within the subscript portion of the argument.

[††]X = legal combination.

CHAPTER 6

THE GLOBAL DATA BASE

The MUMPS language allows local data required by a program to be referenced symbolically and space for it to be allocated as needed. Local data is the set of variables established within the domain of a particular program, and available and defined only within that program. The data actually resides within the user partition, and functions as scratch or transient data. Local arrays are treated as if they are intended to be sparse i.e., only subscripts for which data are defined are allocated space. A symbolic variable used in a program may be given either a numerical value or a variable length string value. When it has a string value, only that space required by the string is actually allocated. Thus, for both strings and sparse arrays, the overhead of a compiler system does not exist (i.e., maximum sizes of arrays and maximum lengths for string variables are not allocated).

This philosophy is extended to data management on disk. Elements stored in data files are referenced entirely symbolically; the file name is similar to that of a local variable name in a program. Fields in the data file are treated as array elements and referenced by means of subscripts; subfields are referenced by appending additional subscripts. Data files on the disk thus comprise an external system of arrays, which provides a common data base available to all programs. The arrays that make up this external system are called global variables, and are identified by global array names. A global name (or file name) consists of the character up-arrow (↑) followed by one to three alphabetic characters. The form of the subscript portion of an array reference consists of an arbitrary number of numeric expressions separated by commas and enclosed by parentheses.

Like local arrays, the range of legal subscript values for global arrays is 0.00 to +327.67. Use of values outside this range will be treated as an error (MAXIM). Each addressable element of a global array is called a <u>node</u>. Thus, ↑A(1), ↑GHC(7,3,4), and ↑B(1,2,9,3,7.46) are all nodes. The structure of these arrays is hierarchical, and any node within an array tree may possess either a numeric or a string data value and/or a pointer to a lower level in the tree. Data may be stored at any level, and there are no constraints to the dimension or size of the array. In addition, the quantity and magnitude of subscripts for an array are dynamic so that both the content of an array and its structure may vary.

6.1 USE OF OPEN AND CLOSE COMMANDS

To avoid time-sharing conflicts, a program must use the OPEN command to prevent other programs from changing global arrays which it is in the process of altering. The argument of OPEN may be one array name or a list of array names. For example:

```
OPEN ↑G
```

or

```
OPEN ↑A,↑B,↑C
```

OPEN prevents any other program from altering data in any of the specified arrays. The effect of OPEN is cancelled when the program halts or at the occurrence of the command CLOSE, which does not allow any arguments, and releases all opened arrays to other users in the system. If an OPEN is unsuccessful, all open arrays will be CLOSEd, and MUMPS continues trying to open the arrays until it is successful. Also, a program error will CLOSE all globals. A program may not alter a global array it has not OPENed.

If the CLOSE command is combined with other commands on the same line, C must be followed by a double space: e.g. >C␣␣G␣1. If a single space were used, MUMPS would attempt a "CALL G".

Since modification of content and structure of a global array may be caused by a variety of programs in the system, a particular program must sometimes examine the current configuration of an array before attempting to access or update it. MUMPS provides a set of functions to determine the type and structure of a global array. These functions ($D and $H, described in Chapter 5) permit the programmer to locate the nodes where information is stored within an array as well as empty nodes that are available for data storage.

6.2 USE OF THE $DEFINE FUNCTION

The program may determine the data type of a specified node by application of the function $DEFINE as $DEFINE(↑ARRAY(NVE1,NVE2,...)). The numeric value returned represents the current status as follows:

Ø - undefined with no datum or structure associated with it
1 - string valued datum only
2 - numeric valued datum only

3 - not used

4 - pointer to structure at lower level only

5 - pointer plus string valued datum

6 - pointer plus numeric valued datum

6.3 USE OF THE $HIGH FUNCTION

The program may determine the next sequential node at a given level by means of the function $HIGH as in $HIGH(↑ARRAY(NVE1,NVE2,...,NVEn)). The function returns the lowest subscript value which is on the same level as NVEn and which is higher than NVEn. To determine the exact state of that defined node the $DEFINE function must be used. If, upon completing the scan of a file, no subscript value greater than that specified is found, $HIGH returns a value of zero (Ø). Thus, when $HIGH is used as an instruction in an iterative process, the value returned must always be checked for zero as an escape. Consider the following example where ↑A consists of nodes ↑A(1), ↑A(1,1), ↑A(1,2), and ↑A(1,1,2).

```
>T $H(↑A(1,1))
2
>T $H(↑A(1,1,1))
2
>T $H(↑A(1,2))
Ø
>
```

6.4 STORAGE OF DATA IN GLOBAL ARRAYS

The storage of data into an array is accomplished solely by the assignment command, SET (i.e., READ and ASK may not be used to directly store data in a global array). Consider the following statement:

```
SET ↑APR(UN,NAME)="JOHN DOE",↑APR(UN,AGE)=34
```

Assume the global array name ↑APR is reserved as an active patient record file. Each patient entry in the file is accessed through his hospital unit number, in this case a local variable UN. Both NAME and AGE are also local variables whose values indicate the appropriate category at the second level of the array. This statement then assigns the string value "JOHN DOE" and the numeric value 34 to the specific second level categories, name and age respectively. Subsequently, a statement such as:

```
SET ↑APR(UN,CHEM,N)=$DATE.",".TEST
```

might define the N^th laboratory test in the Chemistry Lab with the double field entry of the date concatenated with the test name.

6.5 RETRIEVAL OF DATA FROM GLOBAL ARRAYS

Retrieving data from global arrays is no different from retrieving data from local arrays. The statement:

```
TYPE "THE AGE OF ",↑APR(UN,NAME)," IS ",↑APR(UN,AGE)
```

could result in outputting the following.

```
THE AGE OF JOHN DOE IS 34
```

To print out a list of a patient's laboratory tests (assuming ↑APR(UN,CHEM) is the total number of tests defined), the following statement might be used:

```
FOR I=1:1:↑APR(UN,CHEM) T !↑APR(UN,CHEM,I)
```

The KILL command when applied to a specific node in a global array, prunes the array tree at that node. Any data value and/or array pointers to lower level nodes are removed, and that node reverts back to an undefined status. The statement KILL ↑APR(UN) would delete all information for the patient defined by the local variable UN.

6.6 NAKED GLOBAL VARIABLES

Included in the global array syntax is the "naked" global variable. The form of the naked variable consists of the up-arrow followed by a subscript (or a series of subscripts) enclosed in parentheses. This notation is equivalent to the last previously used global array reference, except that the value of the last referenced subscript is replaced with the value of the first subscript in the naked variable. For example, the statement:

```
TYPE " THE AGE OF ",↑APR(UN,NAME)," IS ",↑(AGE)
```

is equivalent to the previous example that typed NAME and AGE.

Once a block of data accommodating a single level of subscripting is referenced, it is maintained in core memory until a reference is given to a different level by the program. Use of the naked variable then permits other data at the same level to be referenced merely by specifying a terminal subscript, so that once a level is reached often no further disk access need be made to manipulate associated information. If any data in a block is altered, it is only written back on the disk when a reference is made to a block other than the one that is in core memory, when a CLOSE or HALT command is given, or when an error causes a program to crash.

By far the most common errors that occur in the use of global arrays stem from the use of the so-called "naked" variable syntax. The following examples illustrate some of the problems that may be encountered. These examples represent only a few of the many possibilities for producing erroneous results when using naked variables. It is a powerful syntax but one that must be used cautiously.

Example A.

```
IF $D(↑A(I,J))=0 SET ↑(J)=VALUE
```

In this case, the user is testing the status of ↑A(I,J) by means of the $DEFINE function. If $DEFINE returns a zero value, that node is undefined. The user then reasons that since $DEFINE has brought him to the desired level, he is safe in using the naked variable. Incorrect!! The user has no way of knowing where the search has ended and thus cannot know the current level. For example, the search may have ended at the first level if no ↑A(I) node was defined. To be safe, the user should spell out the full name as in:

```
IF $D(↑A(I,J))=0 SET ↑A(I,J)=VALUE
```

Of course, if $DEFINE returns a non-zero result, the use of naked variables is perfectly safe.

Example B.

```
OPEN ↑G SET ↑(I)=VALUE
```

To avoid time sharing conflicts, the first global reference after an OPEN may not be a naked reference.

Example C.

```
SET ↑G(I,J,K)=↑(K)+1
```

This command string is legal, but the result may not be what the user expects. The problem lies in the order of evaluation that MUMPS uses for processing a SET command string. The first side of the equal sign (=) that is evaluated is the right side. The use of the naked variable in this case will reference the last used level, not necessarily the same as ↑G(I,J,K) on the left side of the equal sign.

A more proper use would be:

```
SET ↑(K)=↑G(I,J,K)+1
```

This command string will cause the appropriate node to be incremented by one.

Example D.

```
FOR I=1:1:N SET ↑(I,D)=VAL(I)
```

While this example seems benign enough, it is not. Each time MUMPS goes through the FOR loop, it will go down another level in the global array!!! The referenced nodes will look like:

```
...1,D
...1,2,D
...1,2,3,D
...1,2,3,4,D
...1,2,3,4,5,D
etc.
```

6.7 THE JOIN COMMAND

Frequently, nodes may share data at a lower level. For example, there may be a node for each of the proprietary names of a particular job while lower-level data remains the same in each case. The JOIN command can be used to allow nodes to "share" the same data at a lower level. For example:

```
JOIN ↑A(1,3,2)←↑A(7,5,6,4)
```

"joins" the A(1,3,2) node to the A(7,5,6,4) node; as a result, the statement

```
TYPE ↑A(1,3,2,5)
```

will actually output ↑A(7,5,6,4,5).

There are presently several implementation restrictions on use of the JOIN command. In particular, JOINs can only be made within a single global array (i.e., the names must be the same). Also, the node on the left may not have a pointer to a lower level.

CHAPTER 7

PROGRAM AND DATA PROTECTION SCHEME

Program and data file protection is an important part of the MUMPS system. Through the use of the MUPAK utility package, the system manager can control every aspect of application program and data filing and retrieval. Programs are protected from unauthorized use, inspection, or modification by a sign-in scheme involving user name codes, program name codes, and program access keys. Global data is protected by access keys that prevent unauthorized reading or data modification.

MUMPS and MUPAK together provide a comprehensive protection scheme, but the number of protection features that are actually applied may be determined by the installation manager. Some installations may require absolutely no protection -- others may require considerable protection due to their public nature.

Most of the protection scheme is established through the MUPAK utility package. Program protection keys, however, are assigned by the MUMPS program author when the program is filed.

7.1 DEFINITIONS

Some of the terms that describe users, terminals, programs, and data files in the MUMPS system are defined below.

7.1.1 Mode of User Interaction

Direct: In direct mode, the user can enter MUMPS commands for immediate execution, or compose and enter numbered MUMPS program steps. Direct mode is available only at terminals that are assigned developmental status (defined below). Different log-in procedures give the user scratchpad privileges, permission to create new programs, or permission to use or modify existing programs.

Indirect: Indirect mode is in effect while stored program steps are being executed sequentially by MUMPS. The program may type out conversational queries and allow the user to enter data, but the user cannot enter direct MUMPS commands or compose program steps.

7.1.2 Program Status (D or I)

Programs are assigned D or I status by log-in procedures. The MUPAK utility package can also assign certain I-status programs to operate as "autoload" programs for specified users.

D-Status (Developmental): D-status programs may be requested by name during log-in at a developmental terminal. If a D-status program is requested during log-in at an operational terminal, MUMPS prints the message NOT FOUND.

I-status (Independent): I-status programs may be requested by name at operational as well as at developmental terminals. D-status programs can be converted to I-status by a special log-in procedure.

Load and Go: All I-status programs are considered "load and go". When a user logs in in I-mode and requests a library program by name, the program begins execution automatically at the lowest non-zero step number. The terminal remains in I-mode, so the user cannot stop execution and examine or modify the program itself.

Autoload: An autoload program is an I-status program that has been assigned by MUPAK to begin execution when a particular user signs in. The user does not specify a program name. An autoload program can also be assigned to begin execution in response to the operation of the BREAK key at a "tied" terminal.

7.1.3 Terminal Privileges

Each MUMPS terminal can be assigned by the MUPAK utility package to serve in one of four privilege categories -- developmental, operational, tied, or closed. Developmental terminals have unrestricted privileges but log-in is elaborate. Other terminal classes have progressively fewer privileges but simpler log-in procedures.

Developmental: Developmental terminals are allowed access to direct mode for execution of direct MUMPS commands and composition and filing of MUMPS applications programs. Other programs of D or I status can be brought in for execution or inspection in direct mode. A developmental terminal can also execute D- or I-status programs in indirect mode. Direct or indirect mode is established at the time of log-in, not by the type of program that is in effect.

Operational: Operational terminals are restricted to I-status programs. Through specific log-in procedures, the user can request load-and-go programs by name, or begin execution of an "autoload" program associated with his user name code.

Tied: Tied terminals are restricted to a single autoload program that begins execution when the BREAK key is pressed. No other log-in is necessary.

Closed: Closed terminals are the most restricted from the user's standpoint, in that he cannot log in or otherwise request service However, closed terminals can be controlled by a MUMPS application program that prints out instructions and requests information input to be entered at the keyboard.

7.1.4 Protection Keys

A key is a positive MUMPS number in the range 0.01 through 1310.71 inclusive. If the key associated with any key-protected item is zero, then the item is considered to be unprotected. All previous descriptions in the body of this manual have assumed no protection for the sake of simplicity.

7.2 DEVELOPING, FILING, AND USING MUMPS PROGRAMS

MUMPS applications programs are developed, debugged, and modified in the D (direct) operating mode at a terminal that has been assigned developmental status. To prevent unauthorized users from filing new programs or modifying existing ones, the user's intent must be established at the time of log-in.

By citing a scratchpad (SP) key during log-in, a user can gain access to a developmental terminal for direct command execution or the creation of short temporary programs.

By citing a nonexistent program key (NE) during log-in, the user can establish a program name, create the program, and file it permanently in the program library. A protection key can be assigned at the time the program is filed.

Users can request an existing program for execution by specifying a program name during log-in. However, access in D-mode to key-protected programs is denied unless the correct key is specified. Thus, programs cannot be examined or modified by unauthorized users. Any program can be executed on request without citing the key, but a protected program cannot be inspected or changed unless the key is cited.

Operational terminals can only request and execute I-status programs. (However, these programs may call or overlay programs which are filed in D-status.)

Codes and symbols used during log-in are defined in Table 7-1. Log-in examples are described in Table 7-2.

7.2.1 Scratchpad Operation

Log-in: Scratchpad log-in permits a user to operate a terminal without filing any programs. Log in as follows:

1. Press the BREAK key. MUMPS responds by typing:

```
MUMPS LINE 1
ID
```

2. Type user name, scratchpad key [in brackets], and an asterisk:

```
MUMPS LINE 1
ID ABC[3.2]*
```

(User input is underlined.)

If no scratchpad key has been assigned, the key number and brackets are omitted and the asterisk is typed after the user name:

```
MUMPS LINE 1
ID ABC*
```

3. Press the ENTER key. (The ENTER key may also be labeled ESC, ALT MODE, or PREFIX, depending on the type of terminal you have.) If your ID codes are valid, MUMPS responds with a CRLF (carriage return-line feed) and types a right caret:

```
MUMPS LINE 1
ID ABC[3.2]*
>
```

The right caret indicates that the terminal is in direct mode and will accept direct MUMPS commands or indirect step entries.

Scratchpad Capabilities: The commands, example programs, and demonstrations appearing in the body of this manual assume that the terminal is in "scratchpad" mode of operation; the terminal is in direct mode and the user is able to execute direct commands, create new program steps, execute these programs, or

Table 7-1

Log-In Conventions

Item	Example	Explanation
User Name	ABE	Established through MUPAK (usually three letters).
Program Name	ABE:TIM	User ABE's program TIM. (Program name, following colon, can be up to three letters. Numbers are not permitted.)
Mode Designator	*	Asterisk specifies log-in to direct mode. If asterisk is omitted, only I-Status programs can be invoked.
Scratchpad Key	[SP]	Authorizes use of a terminal in scratchpad mode. Actual key is assigned through MUPAK.
Nonexistent Program Key	[NE]	Authorizes use of a terminal in creating and filing a new program. This key is also assigned through MUPAK.
Protection Key	[N.NN]	Authorizes a D-mode user to examine or modify a protected program. Keys are assigned to programs by the user when the program is filed.

NOTE

Keys are positive MUMPS numbers (0.01 through 1310.71).

Table 7-2

Log-In Examples

```
(BREAK)
ID STE[3.25]*
>
```

User STE logs in for a session of D-mode scratchpad operation. The [SP] key can be omitted if set to 0.00 by MUPAK.

```
(BREAK)
MUMPS LINE 1
ID STE:TIM[2.5Ø]* NEW D
>
```

User STE logs in to create program STE:TIM. MUMPS responds by authorizing the NEW program and classifying it in D-status. The [NE] key can be omitted if set to 0.00 by MUPAK.

```
(BREAK)
MUMPS LINE 1
ID STE
THE TIME IS 7:55
THANK YOU
```

User STE logs in for his autoload program, which executes and terminates automatically. (MUPAK assigns the Program STE:TIM as STE's autoload program.)

```
(BREAK)
MUMPS LINE 1
ID STE:TIM
THE TIME IS 7:56
THANK YOU
```

Any user logs in in I-mode for the STE:TIM I-status program. The program operates as a load-and-go (executes automatically and terminates).

```
(BREAK)
MUMPS LINE 1
ID STE:TIM[37.2]*I
>W

1.1Ø T "THE TIME IS "
1.15 S HR=$I($T/36)
1.2Ø S MIN=$I(($T-HR*36)*1Ø/6)
1.25 T HR,":"
1.3Ø I MIN<1Ø T "Ø",MIN Q
1.35 T MIN Q

>D 1
THE TIME IS 8:Ø1
>
```

User STE logs in for his I-status program STE:TIM, specifies the protection key, and requests D-mode (*). Note that MUMPS prints out the filed status of the program (I). The program enters his user partition where it can be run, inspected, or modified.

call and overlay other filed programs. While programs are being executed automatically by MUMPS, the "indirect" mode is in effect: everything that takes place is under control of the program. (To interrupt execution of a long or repetitive program, press the BREAK key. The terminal will stop executing and return to the direct mode.)

In the scratchpad mode, programs cannot be filed, since the log-in procedure does not specify a program name. When a session at a terminal is ended by a HALT command, the programs and any associated local variables in the user partition are erased. Even if the same user logs in again in scratchpad mode, the partition is empty.

Scratchpad-mode programs can be saved on paper tape or DECtape, as described in Chapter 4. When the user logs in again he can read in the same programs, resume development, and execute them.

Call and Overlay Commands: A scratchpad user can employ the CALL and OVERLAY commands to bring in filed programs for execution. However, an unfiled developmental program in the user partition will be lost when the CALL or OVERLAY takes place. MUMPS will attempt to return to the developmental program, but will not find it in the file. An OVERLAYed program simply remains in the partition after it has been completed. No attempt is made to retrieve the original program.

The effects of CALLing or OVERLAYing key-protected programs in D-mode are discussed later.

7.2.2 Creating and Filing New Programs

Log-in: The intent to create and file a new program must be declared at the time of log-in. After the BREAK key is pressed and MUMPS responds with the request for ID, type the user name, a colon, the proposed program name (also one to three letters), the NE key, and an asterisk followed by (ALT). For example:

```
MUMPS LINE 1
ID STE:GNG[2.50]*
```

If the user name is accepted, the program name is legal, and the key is valid, the system responds "NEW D" and enters direct mode (indicated by a right caret):

```
MUMPS LINE 1
ID STE:GNG[2.5Ø]* NEW D
>
```

The "NEW D" response indicates permission to develop a program of the specified name, and that the program is classified as a developmental program (D-mode) (the log-in method for changing the program to I-mode appears later).

Filing a New Program: Once the program has been composed and debugged to the user's satisfaction, it can be stored using the FILE command (usable only in direct mode), for example:

```
>F
```

The program is filed under the user name and program name specified at log-in.

Assigning Protection Key: Protection keys prevent other D-mode users from examining, altering, or deleting filed programs. A protection key may be assigned at the time the file command is given, for example:

```
>F [2.5]
```

A new program initially has a key of zero (no protection). If no key is assigned, it remains zero.

Changing and Re-Filing: After a program has been filed, it remains in the user partition until the use of the terminal is HALTed. The user can continue to run it, or even modify it and re-file. Every time a FILE command is entered, the program currently present in the user partition replaces the previous program in the file. If the new program has fewer steps, unused steps of the old program are erased. However, a program must contain at least one step in order for the FILE command to take effect.

Once a key has been assigned, it need not be repeated. The FILE command leaves the previous key intact. To change a key, simply use a FILE command with the new key specified. To remove a key from a previously protected program, use:

```
>F [Ø]
```

Using Calls and Overlays: While developing a program, the user may want to CALL or OVERLAY another filed program. The developmental program must be filed before a CALL or OVERLAY command is entered. Otherwise, MUMPS will be unable to retrieve the original program.

To CALL or OVERLAY programs filed under another user name, the form ABC:DEF (user name:program name) may be used. For example, the command:

```
>CALL A,SAM:CHG
```

CALLs current user's program A and user SAM's program CHG. If no user name is given, MUMPS assumes that the user name is the last one specified by a CALL or OVERLAY. In the example above, if the program SAM:CHG contains the statement

```
>CALL DJB
```

SAM's program DJB will be CALLed. The current user name is stored on each CALL and restored on return.

Referencing Key-Protected Programs: If a program referenced by CALL or OVERLAY is protected, the key may be cited either in %K or in the command itself:

```
>CALL ROG:ERT[19.2]
```

In a CALL to a protected program, citing the key allows the user to inspect the CALLed program in direct mode if there is an error or interrupt. If a program CALLs a protected program without citing the key, the protected program will run in load-and-go fashion, but the terminal cannot be returned to direct mode until the protected program is done and an unprotected program level is reached. If there is an error, or if the user attempts to interrupt execution with the BREAK key, MUMPS will erase the partition and return to direct mode.

Every OVERLAY attempt in direct mode is by definition an intent to "peek" at the program, since the OVERLAYing program is loaded into the partition and the terminal is returned to direct mode. Therefore, the key of a protected program must be cited in an OVERLAY. If the user attempts to OVERLAY to a protected program without citing the key, the partition is erased and a ?KEY error message is printed.

Programs intended for execution in I-mode may themselves CALL or OVERLAY D-status or I-status programs without specifying keys, even though these programs are key-protected. Since an operational terminal user cannot stop execution and take control in direct mode, keys can be disregarded. Keys prevent programs from being inspected by unauthorized users.

Notes on Keys:

1. Where a key is not required (the key is zero) citing a key is not an error. Thus %K need never be cleared. (%K is set to 0.00 during log-in.)

2. A key cited syntactically (within brackets) takes precedence over the %K value.

3. Although I-mode programs are not protected from being run, the program may contain an internal "password" scheme to prevent unauthorized use.

7.2.3 Modifying Existing Programs

Once a program is created and filed, the originator can run it and modify or update it. So can any other D-mode user, if he knows the program name and protection key.

Log-In: To obtain an existing program and retain control in D-mode, the user must specify the program name and protection key, if any, at the time of log-in:

```
BREAK
MUMPS LINE 1
ID STE:A[12.5]*D
>
```

If MUMPS finds a program of that name and the protection key is correct, MUMPS prints the program status (D or I) and the caret indicating that the program is present in the user partition.

Changing a Program and Re-Filing: The user can execute the program and add, modify, or delete steps just as when developing new programs. A new version can be filed using the FILE command. A new protection key can be assigned when the program is re-filed. (If no key is specified, the previous key stays in effect.)

IN USE Condition: In order to prevent time-sharing conflicts, a program is considered to be IN USE as soon as any terminal logs in specifying that program and getting control in direct mode. Until that terminal logs out no other terminal may log in for that program in direct mode. Thus, it is impossible for two terminals to simultaneously change and file the same program.

7.2.4 Changing Program Status (D, I, and X)

Programs are assigned D(developmental) status when they are created. D-status programs cannot be invoked for execution by I-mode users. If a user attempts to log in for use of a D-mode program in I-mode (i.e., autoload or load-and-go), MUMPS will respond "NOT FOUND".

Changing D to I: In order to release a D-status program to the program library for autoload or load-and-go use, it must be changed to I-status. This is done at log-in time by adding an I after the asterisk in the identification entry:

```
(BREAK)
MUMPS LINE 1
ID STE:A[12.5]*I OK

ID
```

MUMPS types OK, indicating that the named program has been changed to I status, and asks for a new ID.

CAUTION

Before releasing a program in I status, the user should make sure it contains a quit or HALT command or other means of graceful exit. Since an I-mode user's ability to interrupt execution depends on the condition of the %E system variable, he may not be able to terminate a program that loops endlessly.

Changing I to D: An I-status program can be changed back to D status (withdrawn from autoload or load-and-go use) by adding a D after the asterisk during log-in.

Expunging (X): Any existing program can be expunged (deleted from the file) at log-in by adding an X after the asterisk:

```
   (BREAK)
MUMPS LINE 1
ID STE:A[12.5]*X OK
ID STE:A[12.5]* NOT FOUND    Program has now been deleted.
ID                           Replying with a null string
THANK YOU                      logs out the terminal.
```

An expunged program disappears from the file. The program name can be used again.

7.2.5 Changing a Program Name

The following procedure changes a program named XYZ:A*D to the name SDH:B*D

1. Log in to create the new program SDH:B*
2. When MUMPS responds

   ```
   NEW D
   ```

   bring in the existing program with a direct OVERLAY command, i.e.:

   ```
   >OVERLAY XYZ:A
   ```

   or, if there is a key:

   ```
   >OVERLAY XYZ:A[13.21]
   ```
3. When MUMPS responds with the right caret, indicating that the called program is in the user partition, enter a FILE command. The program will be filed under the name specified at log-in. Assign a protection key, if desired.
4. Sign off (HALT).
5. Log in again using the name of the old program and add an X to expunge it from the file.

7.2.6 Using Programs in I-Mode

In actual practice, the direct mode is restricted to users who are authorized to create programs and file them. Day-to-day users, at operational terminals, do not compose programs. Such terminals are limited to conversational data entry and message printout under control of an applications program. The terminal never leaves the I-mode to permit direct command insertion or program composition. I-mode users cannot invoke developmental (D-mode) programs by name for execution.

Log-In: I-mode users may obtain programs in three ways: by user name/program name (load-and-go), by user name only (autoload), or by operation of the BREAK key (at tied terminals). I-mode log-in codes (user names and keys) are controlled through MUPAK by the MUMPS system manager. If the name and key given at log-in are legal, MUMPS loads the program and begins execution at the first non-zero step. The terminal is never released to direct control. The result of an error or an attempt to stop the program by pressing the BREAK key depends on the value of the %E system variable (Table 4-1). When the program runs out of steps, or reaches a quit or halt command, MUMPS terminates the session at the terminal. To obtain another program, the user must log in again.

Requesting Programs by Name (Load-and-Go): An I-mode user signs in for a specific I-mode program by entering user name and program name. The asterisk is not used:

```
MUMPS LINE 1
ID STE:TIM
```

Since I-mode users cannot inspect programs, the key numerals (if assigned) and brackets are omitted.

Autoload Program: Any I-mode program may be assigned by MUPAK to a particular user name as that user's "autoload" program. For example, program ABE:TIM[37.3]I could be assigned as the autoload program for user ABE. User ABE only needs to enter his user name in order to obtain the program, which begins immediate execution. The same I-status program may be run by anyone who enters the proper user name and program name at log-in without citing a key; since direct mode cannot be entered, the program cannot be examined or changed. An I-status program may always be loaded via autoload or load-and-go even if it is IN USE or other terminals have invoked it via autoload or load-and-go.

Tied Terminals: MUPAK can also assign an I-mode program to act as the autoload program for a specific tied terminal. The program will load and begin execution in response to an operation of the BREAK key; no log-in is required. For example, program SDH:A[3.2]I might be assigned as the autoload program for terminal 5. Whenever the terminal 5 BREAK key is pressed, that program begins execution.

End Use of I-Status Programs: The purpose of MUMPS application programming is to develop a set of programs useful to the ultimate user (e.g., the nurse, clerk, etc.) These users must be given entry to the system through passwords used during log-in. These passwords are usually names of programs filed in the I-status.

A practical application program structure always contains a set of programs and subroutines calling and overlaying each other. The bulk of these routines should remain in D-status to preclude inadvertent sign-in to a subroutine or a piece of the overall package in the wrong sequence.

It is possible that all users may sign in for the same root program, or sub-executive, which routes the user to appropriate D-status pieces of the package. Thus, the only program in the system to be filed in I-status might be that root program.

Hence the guideline: the only program(s) that must be filed in I-status are autoload programs and those which must be known by name by the ultimate users during log-in. If such programs are not I-status, the error message "NOT FOUND" will be given in response to log-in.

7.2.7 Summary of Program Protection Rules

1. The status of a program (i.e. Independent or Developmental) only has meaning at log-in time. In particular, the program status determines the action in case of error or BREAK key operation. If the log-in command string does not include the asterisk, the program must be filed in I-status or execution will not begin. If there is an error or the BREAK key is pressed, %E will be interpreted according to Table 4-1. If log-in does include the asterisk, the program may be filed in either I or D status. If there is an error or the BREAK key is pressed, %E will be ignored and control will revert to the user in direct mode.

2. Assume a user has logged in for development. As he composes his program he may invoke any program he knows by name, using CALL and OVERLAY commands, and they will execute. However, he will not be allowed inspection privileges unless he also cited the program's key. (Program protection keys only have to do with inspection privileges, not execution.)

3. The purpose of operational and tied terminals is to absolutely preclude development from those terminals.

4. The purpose of closed terminals is to absolutely preclude any log-in procedure.

7.3 PROTECTION OF GLOBALS

A global may be key-protected to prevent unauthorized examination and/or alteration of global data. A global has two keys in order that authorization can be given to examine data without implying authorization to change it. Since a global must be open in order to change it, these keys are known as the READ KEY and the OPEN KEY. Since altering a global implies reading (or examining) it, the OPEN key is sufficient for any legal operation on the global. Both keys are set by the MUMPS Utility Package (MUPAK).

7.3.1 OPEN Key

A key protected global may be opened either by loading the key into the system variable %K, or by using the key within the syntax of the OPEN command. For example, to open global ABC which has an OPEN key of 1.27, either of the following will suffice:

```
              S %K=1.27 O ↑ABC
          or
              O ↑ABC[1.27]
```

The second form is necessary when it is required to open more than one key-protected global in a single statement; for example:

```
          O ↑ABC[1.27], ↑BCA[7.21]
```

Once a global is open, no key need be cited to perform any legal operation on that global.

If the global has an OPEN key value of Ø, no key need be cited in order to open the global (the global is considered to be unprotected).

7.3.2 READ Key

The READ key is used to enable examination of a global by a user who is not authorized to alter it (does not have the OPEN key). For example, where ABC had a READ key of 99, either of the following will suffice:

```
              S  %K=99,A= $D( ↑ABC(1,2))
      or
              S  A=$D( ↑ABC[99](1,2))
```

As with the OPEN key, the second form is useful where reference is made to more than one global; for example:

```
                S  A= ↑ABC[99]( ↑X[1](4))
        or
                 S  A= ↑ABC[99](3),B= ↑X[1](4)
```

However, if ↑ABC and ↑X were OPEN in the above examples, the keys would not be required.

Since the OPEN key implies permission to read, it may be used wherever the READ key would otherwise be required.

APPENDIX A

EXPLANATION OF MUMPS MESSAGES

When execution of a MUMPS program is terminated by an error or a break, the program executive outputs a short message to indicate the reason for termination. This message is preceded by the number of the Step being executed unless the error occurred while in Direct mode. It is useful to categorize the types of messages which may be printed:

1. MUMPS programming error messages

   This class of message results from errors associated with programming problems (either in the language syntax or semantic misunderstandings).

2. Operating System error messages

   This class results from various troubles which are detected by the operating system and which are beyond the control of the MUMPS application programmer.

3. Voluntary program termination message

   There is only one message of this type, and it is

   ?IOINT

   indicating the Break key has been activated for a voluntary termination of execution.

4. Debugging Aid message

   The ?BREAK message is an indication that a BREAK command has been encountered in the program.

All errors are considered terminal. After encountering any error, it is not possible to resume execution of the program from the point of error.

However, the BREAK decoding aid is associated with a GO command which permits execution to resume at the step following the BREAK.

Each of the MUMPS messages is explained below.

MUMPS PROGRAMMING ERROR MESSAGES

Message	Explanation
CLOSE	An attempt has been made to store into an unopened Global.
CMMND	Indicates illegal use of a command. This includes using a command that normally takes an argument and omitting the argument, as well as using a command that has not been defined in the language.
DISK!	Hardware error encountered during disk transfer.

FRACT	Indicates that a fractional number was encountered when the process being executed was expecting an integer number. Also invoked when a step number has no fractional part.
FUNCT	Indicates illegal use of a function. Includes the use of an undefined function as well as an illegal use of an argument for a legal function.
GLOBE	Improper Global access.
IOLOK	A reference has been made to a device that is not "owned" by this partition.
KEY	An attempt has been made to perform an operation without citing the required key.
MAXIM	Indicates that the value of a number has exceeded the positive bounds set by the MUMPS system. The maximum value for a number is +1310.71. Also used to indicate that a string has exceeded 73 characters.
MINIM	Indicates that a number has too many digits following the decimal point.
MINUS	Indicates that a negative or zero number was encountered when a positive number was expected. For example, MUMPS will cause a MINUS error if the user tries to call a subscripted variable with a negative subscript. Only positive subscripts are allowed.
MIXED	Indicates that the user has mixed string and numeric arguments in the same expression. For example, if the variable NAME contains the text "JOHN DOE" and the variable NUM is equal to the numeric value 3, the expression NAME+NUM would cause a MIXED error.
STACK	Indicates that the available stack space is used up. Generally indicates nesting is too deep in DO, FOR, or CALL statements, or that a GOTO is executed while in the range of a FOR clause.
STEPS	Indicates that user has tried to reference steps that are not defined in the program buffer. Also invoked when an attempt has been made to modify the step currently being executed.
STORE	The amount of free space in the user's partition is too small to allow increased allocation of space to the symbol table or step buffer. The variable $STORAGE contains the amount of free space, in terms of characters, left in the partition buffer. For example, $STORAGE equal to 0.72 indicates that there are seventy-two characters of free space left.
SYMBO	Indicates misuse of a symbol (e.g., a non-subscripted reference to an array).
SYNTX	Indicates that the current step being executed has an error in syntax. Syntax errors include illegal punctuation, illegal use of operators, illegal use of parentheses, as well as errors encountered in editing a step. Syntax errors comprise a great majority of errors made in the MUMPS system and usually the user will be able to determine the exact cause of the error by merely looking at the step concerned.

TRAP	Your terminal has been disabled.
UNDEF	Indicates that the program has tried to reference a variable that is currently undefined.

VOLUNTARY PROGRAM TERMINATION MESSAGE

Message	Explanation
?IOINT	Signifies an attempt to interrupt execution by pressing the break key.

OPERATING SYSTEM ERROR MESSAGES

Message	Explanation
.....	Unspecified error.
BLOCK	The free list is too low to allow setting globals or filing programs. Call the computer operator.
CHPNT	Bad data on the disk; call the computer operator.
IODSK	Disk error.
IODT	DECtape error.
IOERR	Line Printer or Card Reader error.
IOPTP	Paper Tape Punch error.
IOPTR	Paper Tape Reader error.

DEBUGGING AID MESSAGE

Message	Explanation
? N BREAK	Indicates that program control has reached a BREAK command at step N. BREAK commands are used to interrupt execution of the program to allow the user to more easily debug his program.

ERRORS DURING CALL AND OVERLAY

If errors are detected in a program other than the program name (and user name) specified at log-in, because a call or overlay is in effect, the error message is expanded.

Example:

Suppose user JOE logs in for his program AXE, and an error occurs in

JOE:AXE. The printout is:

```
             ?ERROR  2:10      (Error detected during Step 2.10)
```

Now suppose JOE:AXE had been error free, but called JOE:BAD which contained an error. The printout is:

```
             ?ERROR  :BAD  3.20
```

indicating that JOE:BAD step 3.20 caused the error (the "JOE" is suppressed since it is assumed the user remembers what user code he entered at log-in.

Furthermore, suppose program JOE:AXE calls a general system program filed under user code SYS and named GEN, and that an error occurs while it is executing. The printout is:

```
             ?ERROR  SYS:GEN 4.40
```

INDEX

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections are published by Software Information Service in the following newsletters.

Digital Software News for the PDP-8 & PDP-12
Digital Software News for the PDP-11
Digital Software News for the PDP-9/15 Family

These newsletters contain information applicable to software available from Digital's Program Library, Articles in Digital Software News update the cumulative Software Performance Summary which is contained in each basic kit of system software for new computers. To assure that the monthly Digital Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest Digital office.

Questions or problems concerning Digital's Software should be reported to the Software Specialist. In cases where no Software Specialist is available, please send a Software Performance Report form with details of the problem to:

Software Information Service
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

These forms which are provided in the software kit should be fully filled out and accompanied by teletype output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software and manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest Digital Field office or representative. U.S.A. customers may order directly from the Program Library in Maynard. When ordering, include the code number and a brief description of the software requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information please write to:

DECUS
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

M U M P S
Programming Language
DEC-15-GXZC-D

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback -- your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability and readability.

Did you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Please state your position. ______________ Date: ______

Name: ______________ Organization: ______________

Street: ______________ Department: ______________

City: ______________ State: ______________ Zip or Country ______

------------------- Fold Here -------------------

------------------- Do Not Tear - Fold Here and Staple -------------------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Digital Equipment Corporation
Software Information Services
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754