

pdp11

**INTRODUCTION  
TO  
MUMPS-11 LANGUAGE**

Order No. DEC-11-MMLTA-C-D

digital

**INTRODUCTION  
TO  
MUMPS-11 LANGUAGE**

Order No. DEC-11-MMLTA-C-D

First Printing, June 1973  
Revised: May 1974  
July 1974  
January 1976

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1973, 1974, 1976 by Digital Equipment Corporation

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECTape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECsystem-20	TYPESET-11

## ACKNOWLEDGMENT

MUMPS-11 is an integrated system comprised of an interactive programming language, a data management facility and a multiuser time sharing executive, developed by Digital Equipment Corporation for the PDP-11. Meditech Corporation contributed to the original development of MUMPS-11 and Interpretive Data Systems Inc., assisted in the Version 4 developments to MUMPS-11.

The language is a dialect of MUMPS (Massachusetts General Hospital Utility Multi-Programming System) which was developed at the Laboratory of Computer Science at Massachusetts General Hospital and is supported by Grant HS00240 from the National Center for Health Services Research and Development.



# CONTENTS

	<u>Page</u>
PREFACE	vii
CHAPTER 1    SAY SOMETHING IN MUMPS	1-1
CHAPTER 2    COMPUTER POWER	2-1
CHAPTER 3    MAKING DECISIONS	3-1
CHAPTER 4    CONSERVATION OF ENERGY	4-1
CHAPTER 5    GETTING IT TOGETHER	5-1
CHAPTER 6    FORM FOLLOWS FUNCTION	6-1
CHAPTER 7    INSIDE GLOBALS	7-1



## PREFACE

*Introduction to MUMPS-11 Language* is a tutorial manual that introduces you to the basic elements and concepts of the MUMPS-11 language. Its intent is to familiarize you with MUMPS-11, rather than to provide comprehensive reference data.

We have organized the manual in a serial fashion -- each succeeding section and chapter building on previously presented information. Chapter 1 tells you how to begin using MUMPS-11 at a terminal. The remaining chapters present the language in the context of a hypothetical census data gathering application. New ideas are introduced to you as required to develop the application programs.

If you're a novice programmer, try to use this manual in conjunction with a MUMPS-11 terminal so that you can do the many examples we've provided.

If you're an experienced programmer, you'll probably be more interested in the reference documentation listed below. However, you may want to skim the first five chapters to get the "sense" of MUMPS-11. The remaining chapters (6 and 7) deal extensively with MUMPS-11 data structure; you should read them carefully.

In the back of the manual is a glossary of terms peculiar to MUMPS-11. You should find this helpful.

When you've finished with this manual, you'll want to read the reference documentation listed below for comprehensive information about MUMPS-11 language, programming and operating procedures. All of the manuals also contain a common set of appendices covering: Glossary of Terms, Character Set, Error Messages, Symbol Usage, and Conversion Tables.

Differences between this manual and the previous revision are indicated by a heavy solid line adjacent to the affected area in the outer margin of the page.



- o *MUMPS-11 Language Reference Manual* - DEC-11-MMLMA-C-D
  - Elements of the language: the character set, programming modes, program structure, data modes, numbers, strings, literals, constants, and variables.
  - Expressions: how to form them and how they are evaluated.
  - MUMPS-11 Commands: meaning, syntax, arguments, and examples of use.
  - MUMPS-11 Functions: meaning, syntax arguments, and examples of use.
  
- o *MUMPS-11 Programmer's Guide* - DEC-11-MMPGA-C-D
 

This manual provides all information required to create, execute, and save MUMPS-11 programs.

  - System Overview: MUMPS-11 hardware/software environment and functional description of the operating system.
  - Terminal Usage: Log-in/log-out procedures, terminal types, special keyboard control characters.
  - Programming Techniques: creating programs, loading programs, storing programs, program size considerations, using system variables, conserving space, ... and more.
  - Using I/O Devices: general concepts of input/output, specific device characteristics.
  - Library Utility Programs: functional characteristics, how to run them.
  
- o *MUMPS-11 Operator's Guide* - DEC-11-MMOPA-D-D
 

This manual contains information for system operators and system managers. Subjects covered include: Operator Controls, Building the System, System Generation, System Operator Functions, Error Detection and Recovery.
  
- o *MUMPS-11 Programmer's Reference Card* - DEC-11-MMPCA-C-C
 

Pocket reference card containing: command and function summaries, messages, symbology, character set, etc.

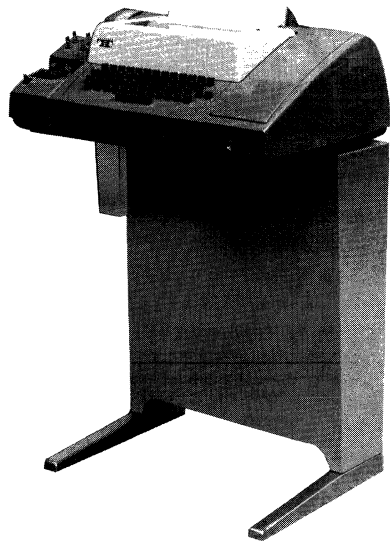
# CHAPTER 1

## SAY SOMETHING IN MUMPS

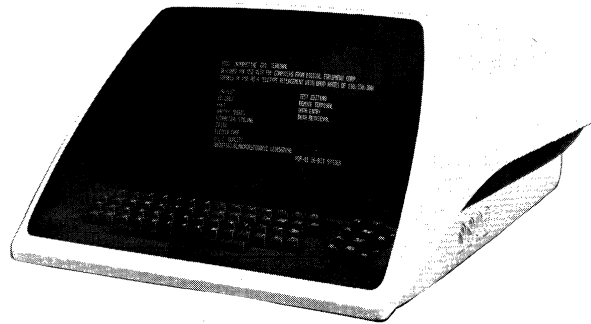
### GETTING STARTED

How do we communicate with MUMPS?

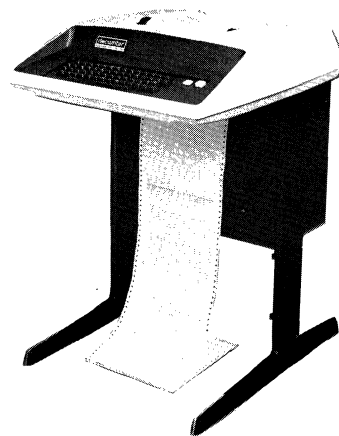
With a terminal, of course. Three of the most common MUMPS terminals are:



the Teletype,<sup>1</sup>



the VT05 Terminal,



and the LA30 Teleprinter.

---

<sup>1</sup>Teletype is a registered trademark of the Teletype Corporation.

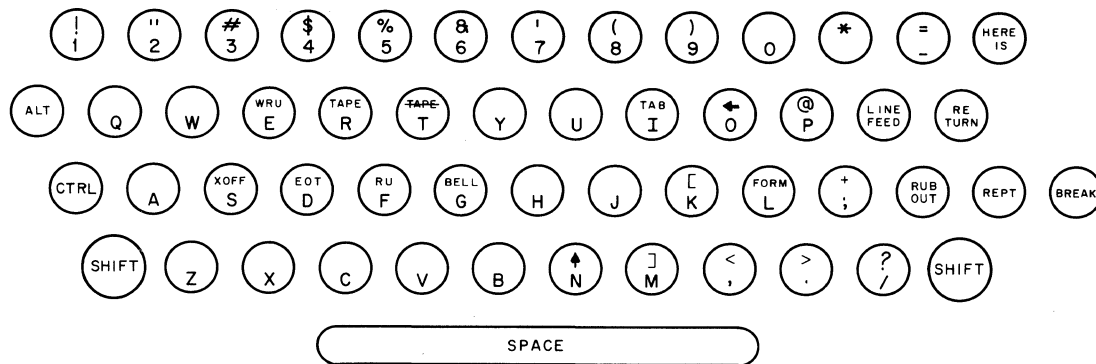
The terminal is connected to the MUMPS computer in one of two ways:

- Hardwired, by means of a cable that directly connects it to the computer.
- Indirectly, using the telephone system as the link. In this case, the terminal is remote from the MUMPS computer...it can be in any place that has telephone service.

We use the terminal to send messages to MUMPS. In turn, MUMPS sends messages to your terminal for you to read.

The messages we send are in a language called MUMPS. This book is designed to help you learn to converse with a computer which uses the MUMPS language.

Let's begin. Here is a diagram of a Teletype keyboard. Look at it closely. If you have a different type terminal you'll notice there are some differences. Don't worry about them... they're not important now.



11-1801

Locate the keys with these characters:

- Letters:            ABCDEFGHIJKLMNOPQRSTUVWXYZ
- Digits:            1234567890
- Symbols:          !"#\$%&'()\*+,@[\+↑]<>?:-;.,./

To type a letter or a digit, simply press the appropriate key.

To type symbols like  `; / . , -`  
press the appropriate key.


To type one of these symbols  `! " # $ % & ' ( ) * = + - @ [ \ + ↑ ] < > ?`  
hold down the SHIFT key  
and press the key with the  
desired character.

*NOTE*

*Some Teletype keyboards may have different characters on some of the keys.*

- *The ALT MODE key may be labelled ESC.*
- *The up-arrow (↑) may be represented as a circumflex (^).*
- *The back-arrow (←) may be represented as an underscore (\_).*
- *The RUBOUT key may be labelled DELETE.*

SMALL TALK

Begin! Type something on the terminal. Press the  key when you are done.  
Nothing happened, did it?

*The point is ... MUMPS  
didn't understand you.*

You have to introduce yourself before beginning a conversation with MUMPS. MUMPS won't talk to strangers. If you type your name, MUMPS won't recognize it. MUMPS knows people by special codes that your System Manager must give you. There are two codes that you need: a User Class Identifier Code and the Programmer Access Code.

*Everyone calls these codes  
by the abbreviations UCI  
and PAC.*

Let's assume your initials are JMW. Your System Manager could use them as your UCI. Further, let's assume the PAC is CTRL X CTRL X CTRL X.

Before you can introduce yourself to MUMPS with your UCI and PAC, you've got to get its attention. Hold down the CTRL key and press the C key.


*You've just typed CTRL C.  
This is the way to get MUMPS'  
attention.*

When MUMPS receives the CTRL C signal, it identifies itself at your terminal by printing a message similar to the following:

MUMPS-11 V03 #6  
UCI:

*This is the particular  
version of MUMPS.*

*This is the number assigned  
to your terminal.*

and then waits for you to answer. MUMPS has a short attention span. So if you don't answer within 20 seconds it types EXIT and forgets about you until you type another CTRL C. Right after MUMPS' UCI request, type your UCI code, followed by a colon (:). Then type the PAC. CTRL X is typed by holding down the CTRL key and pressing the X key. Do this three times. Then press  .

### Convention

When you use a MUMPS-11 terminal, each line that you type must end with the Carriage RETURN line terminator. Since no character is printed when you type RETURN, the examples in the first few chapters use this symbol to help you.

RE-  
TURN

What you typed would look like this (underlined):

```
MUMPS-11 V03 #6  
UCI: JMW:  
>
```

*Notice that the X's weren't printed. Any letter that's typed when the CTRL key is pressed won't be printed. This prevents unauthorized persons from obtaining the PAC.*

If you didn't make any typing errors, MUMPS will indicate its readiness to converse with you by typing a right caret:

```
>
```

and waiting for you to type some command.

*If you made an error, MUMPS types EXIT or some other message, and you'll have to start over again with CTRL C.*

This procedure of introducing yourself to MUMPS is called *logging-in*.

Now that you have logged-in to MUMPS, type something ... anything. Then press the

 key.

MUMPS will probably type:

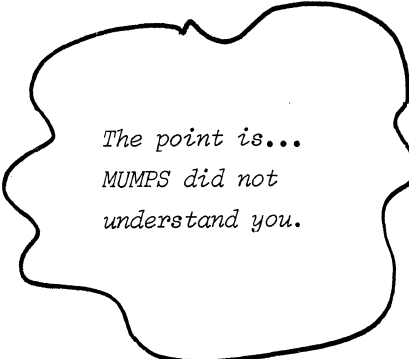
```
SYNTAX>0 @  
>
```

or perhaps:

```
CMMND>0 @  
>
```

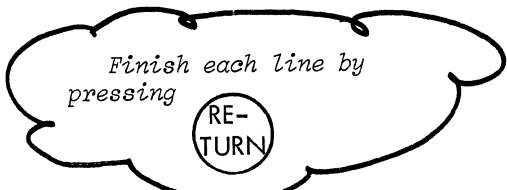
or maybe some other error message:

```
SPNER>0 @  
>
```



*The point is...  
MUMPS did not  
understand you.*

So try this ... you press the keys ... you do the typing.




*Finish each line by  
pressing*



Do it! You type:

```
>ERASE  
>I .I0 TYPE "HELLO, THIS IS MUMPS-11"  
>
```



*one space*

Then type:

>DO, 1

*again - one space*

and MUMPS will type


HELLO, THIS IS MUMPS-11

>

*Nice thought, but it wasn't an original MUMPS idea. MUMPS simply printed what you told it to print.*

Let's review what happened. First you typed:

>ERASE

and pressed the  key. This told MUMPS to ERASE any old program in your program area in memory (your partition). Like erasing a blackboard before writing on it.

Next, you typed a short *program*, consisting of one line with one statement on it.

>1.10 TYPE "HELLO, THIS IS MUMPS-11"

Each line of a MUMPS program begins with a decimal fraction (e.g., 1.1, 1.05, 1.34) followed by one space. Note that numbers like 1.1, 5.7, and 2.3 are equivalent to 1.10, 5.70, and 2.30, respectively.



As you typed the program, MUMPS stored it in its memory. When you finished the line by typing **RE-TURN**, MUMPS answered with its right caret prompting symbol (>) when it was ready for you to type more. After entering the program, you then typed:

```
>DO 1
HELLO, THIS IS MUMPS-11
>
```

*MUMPS ran the program and typed.*

Then it stopped.

The command: **TYPE "HELLO, THIS IS MUMPS-11"**

tells MUMPS to type this message.

Since there were no more command lines higher than 1.1, MUMPS stopped.

Every line in a MUMPS program must begin with a number. These numbers are called step numbers. A step number is a positive decimal number between 0.01 and 327.67. Never use numbers with a zero decimal fraction. Numbers like 1.00, 6.00, etc., are illegal.

When you typed:

```
>DO 1
```

This told MUMPS to DO any steps that have a 1 in the integer part of the step number, like: 1.99 and 1.01 and 1.11, etc.

You can ERASE this program by typing ERASE followed by **RE-TURN**, of course.

ERASE deletes all program steps.

Still your turn. Try this one ... type:

```
>ERASE                                     ERASE the old program.
>1.10 TYPE "7+5"                           Enter the new program.
>1.20 TYPE "ISN'T MUMPS GREAT?"           Here's an additional step.

>DO 1
7+5ISN'T MUMPS GREAT?
>
```

MUMPS types what you tell it to type.

Next ... let's replace step 1.1 with a new step 1.1.

```
>1.10 TYPE 7+5                             No quotation marks.
```

To replace the contents of a step, simply retype the step, putting in the desired commands.

Now tell MUMPS to WRITE out the current program, then DO it.

```
>WRITE
1.10 TYPE 7+5                               Here's the new step 1.1
1.20 TYPE "ISN'T MUMPS GREAT?"           and the old step 1.2.

>DO 1
12 ISN'T MUMPS GREAT?                       DO it.
>                                           This time MUMPS does the
                                           arithmetic, but it's not very
                                           readable.
```

Let's make our output more readable. Change step 1.2 by adding some spaces at the beginning of the message.

```
>1.20 TYPE "   ISN'T MUMPS GREAT?"
```

We added three spaces here

WRITE out the program again to check the change, then try it!

```
>WRITE
1.10 TYPE 7+5                               Old step 1.1
1.20 TYPE "   ISN'T MUMPS GREAT?"         New step 1.2

>DO 1                                       DO it
12   ISN'T MUMPS GREAT?
>
```

The command

```
TYPE 7+5
```

tells MUMPS to evaluate the *arithmetic expression* 7+5 (that is, do the arithmetic) and type the results.


The command

```
TYPE "7+5"
```

tells MUMPS to type the *string* of characters enclosed in quotation marks *exactly as it appears*. No arithmetic is performed.


Strings? Arithmetic expressions?

```
TYPE "ISN'T MUMPS GREAT?"
```




This is a string. It's enclosed in quotation marks!

```
TYPE "7+5"
```



This is also a string.

```
TYPE 7+5
```



This is not a string -- it's an arithmetic expression.

Your turn again. Try this.

```
>ERASE
```

```
>1.1 TYPE "7+5=",7+5
```

```
>DO 1
```

```
7+5=12
```

```
>
```

Note the comma between the "7+5=" and the 7+5.

Replace step 1.1 as follows:

```
>1.1 TYPE "7+5=", " ,7+5
```

We added another string for the TYPE to work on. It's a string of three spaces. Notice the additional comma!

Now WRITE out the modified program and DO it!

```
>W
>I .I TYPE "7+5="," " ,7+5
>DO I
7+5= 12
>
```

If a TYPE command has more than one element (argument), whether string or numeric, each element must be separated by commas.

Try this:

```
>ERASE
>I .I TYPE 7+5,7-5,7*5,7/5
>DO I
122351.40
>
```

Did MUMPS make a mistake? Look closely ... all the answers are there. But they are bunched up. To fix that, we'll put strings of three spaces between the expressions.

Replace I.I, then DO it again!

```
>I .I TYPE 7+5," " ,7-5," " ,7*5," " ,7/5
>DO I
12 2 35 1.40
>
```

*strings of  
three spaces*

To tell MUMPS to add, use +  
To tell MUMPS to subtract, use -  
To tell MUMPS to multiply, use \*  
To tell MUMPS to divide, use /

## MUMPS SHORTHAND

Up till now, the MUMPS commands you've been using were complete words like WRITE, TYPE, DO and ERASE. When you begin to write longer programs, spelling out each of the commands becomes a tedious chore. Fortunately, MUMPS recognizes all its commands by their first letters. It knows that W means WRITE, E means ERASE, T means TYPE, and D means DO. Let's try this shorthand with some of the examples that you've done on the previous pages.

How about

>E	<i>ERASE</i>
>I .1 T "7+5=",7+5	<i>TYPE</i>
>W	<i>WRITE</i>
I .1 0 T "7+5=",7+5	
>D 1	<i>DO</i>
7+5=12	

Now you try some of the previous examples using MUMPS shorthand. Get used to thinking about MUMPS commands by their abbreviations. It's good programming practice to use abbreviated commands ... serious MUMPS programmers always do.

Abbreviating MUMPS commands lets you write programs faster and, more importantly, they take less space in your core memory area (partition). This means you can run larger programs.

Back to more examples.

*We'll use the abbreviated form from now on.*

Mixed operations? Try this one!

```
>I .10 T 2*3+4," ",2*3+4*5," ",2*8/4-2
>D 1
10 50 2
>
```

Were these the answers you expected? Probably not. MUMPS simply performed the operations you specified in left-to-right order, just as it found them. This is the way MUMPS evaluates *expressions*.

Use parentheses to group terms:

```
>I .1 T 2*(3+4)," ",(2+3)*(4+5)," ",(2*8)/(4-2)
>D 1
14 45 8
>
```

MUMPS evaluates all expressions in strict left-to-right sequence. Parentheses must be used to establish any other precedence of operation.

Try some large numbers.

```
>E
>I .10 T 43876543/3.14

>D 1
MXNUM>I .10 @
>
```

What did that message say?

MUMPS found a number that was too large

in Step 1.1

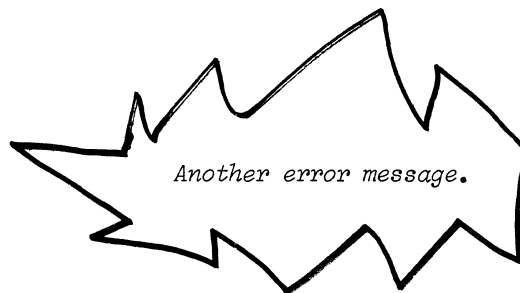
If your step was part of a named program, the program name would appear here. We'll tell you more about this in a later chapter.

Your program didn't work. MUMPS typed the error message MXNUM to tell you that you used a number that was too big.

In MUMPS, numbers are signed, fixed-point quantities in the range  $\pm 21474836.47$

Try some small numbers :

```
>1 .1 T 1.023*100
>D 1
MINIM>1 .10 @
>
```



This program didn't work either. MINIM means that you tried to use a number that had more than two decimal places.

```
1.1 T .05*.5*10
>D 1
0.20
>
```

*Did you expect the answer 0.25?*

This program worked, but you lost part of the answer because your program created an intermediate result that was greater than two decimal places ( $.05 * .5 = 0.025$ )

MUMPS *truncates* the results of all expressions to two decimal places.

How about string arithmetic operations?

```
>I .I T "112"/"12"
```

```
>D 1  
9.33  
>
```

*Don't be surprised at this. Because division is an arithmetic operation, MUMPS simply converted the numeric strings to numbers and performed the division operation.*

Try this one:

```
I .I T "112 INCHES"/"12", " FEET"
```

```
>D 1  
9.33 FEET  
>
```

*MUMPS converted the numeric portions of the strings to numbers and ignored the non-numeric characters.*

When used where numeric values are expected, all leading numeric characters in a string, including +, -, and decimal point (.), are changed to the corresponding numeric value within the range of MUMPS numbers ( $\pm 21474836.47$ ). That is, "112"/"12" is equivalent to 112/12. The first character in the string that does not conform to the format of a MUMPS number terminates the conversion process and the accumulated total is taken as the result. Strings that do not contain leading numeric characters produce a numeric 0.

## MISTAKES

Do you occasionally make mistakes? We do ... watch.

```
>I .I T "7+5="7+5
```

```
>D 1  
SYNTAX>I .I 0 @  
>
```

*We left out the comma between the two arguments.*

*So MUMPS found a syntax error.*



If we had noticed the error before **RE-TURN** was typed, it could have been corrected without having to retype the whole line. There are two ways to correct typing errors when the line isn't terminated by **RE-TURN**.

The **RUB OUT** key lets you delete single characters beginning with the last character that you typed. Each time you press **RUB OUT**, MUMPS prints a backslash (\) so you can tell how many characters have been deleted.

```
>I .I I "7+5="7+5\\\,7+5
>W
I .I I "7+5=",7+5
>
```

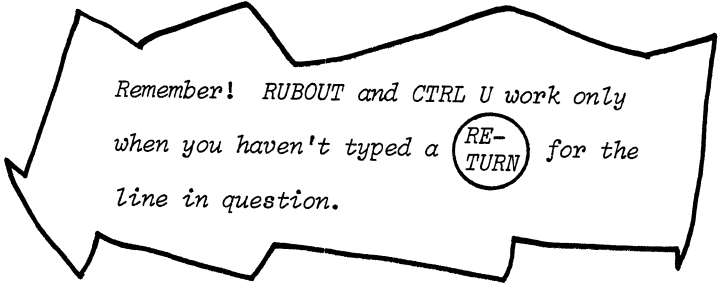
*We type three RUBOUTs to delete the characters up to where we want to insert the comma. Now Write out the step to verify it.*

The CTRL U feature lets you delete the entire line. Simply hold down the CTRL key and press the U key. MUMPS prints an ↑U so you'll know what happened. Try this!

```
>E
>I .I I "ISN'T BUMPS GREAT?" ↑U
>W
>
```

ERASE

*CTRL U wiped out the whole line. Check for yourself. See - it's gone.*



*Remember! RUBOUT and CTRL U work only when you haven't typed a **RE-TURN** for the line in question.*

If we had noticed the error after **RE-TURN** was pressed, we could either retype the step or ERASE it. If we use ERASE, however, all the steps in our program, both good and bad, will be ERASED, unless we tell MUMPS which steps to ERASE. Like this:

>E 1.1 Erases step 1.1 only.

*one space*

>E 1.3,1.8 Erases steps 1.3 and 1.8

*comma*

THE GENERAL FORM OF THE ERASE COMMAND IS:  
ERASE  $n_1, n_2, n_n$   
where  $n_1, n_2, n_n$  are MUMPS step numbers.

Can I do this? Can I do that? What happens if I .....?  
(You complete the question.)

Obviously, we can't answer all your questions in this book. But you and MUMPS can answer most of them.

EXPERIMENT! ... GAMBLE! ... GUESS ...

THEN TRY IT

When you're ready to end your session at the terminal, use the HALT command.

Type:

>H  
EXIT

*MUMPS types this message to tell you that you've been LOGGED-OUT.*

To use the terminal again, just log-in the way we told you at the beginning of this chapter.



## CHAPTER 2

# COMPUTER POWER

Now that you've learned how to converse with MUMPS, let's do something a little more challenging than simple arithmetic. Two important characteristics of MUMPS are that (1) it can perform lots of simple tasks much much faster than people and (2) MUMPS can store tremendous amounts of data.

### THE CENSUS TAKER


With these ideas in mind, let's use MUMPS to help us gather and store the census data of a town. The first thing we have to consider is how to store the data. With MUMPS there are a number of places to store information -- memory, disk, magnetic tape, paper tape. Since memory is the most basic kind of storage medium, we'll talk about it first -- we'll discuss disk storage in later chapters.

If you weren't using MUMPS with your computer, you could only store your data in memory if you knew the addresses of the actual memory locations to be used. This can be a complicated task. When MUMPS is being used, you don't have to worry about the actual location of your data in memory. You can invent your own names for the memory storage location to be used. You can create symbolic names like AGE, SEX, NAM, P, or A3 to define memory storage areas.

A storage area can contain either a number or a string of characters at any given time -- either one or the other -- not both. Since programs often change the data stored in these symbolic memory locations, they are called *variables*. Remember, in the examples in Chapter 1 you stored data directly in the programs, either as numbers (constants) or as strings enclosed in quotation marks (literals).

Remember?

1.1 T 1+2



*constants*

and also

1.2 T "1+2"



*literal*

Each time you wanted to change the data, the program had to be rewritten.

Now we can write programs like:

```
1.1 T A*B
```

Do it -- type it in.

```
>1.1 T A*B
```

Now DO it.

```
>D 1
```

```
UNDEF>1.10 @  
>
```

*Too bad - an error message! MUMPS is telling us that an UNDEFINED variable was referenced.*

In fact, both variable A and variable B are undefined. Before a variable can be interrogated, it must first be SET to some numeric value -- even zero -- or to some string value -- even space.

Type this:

```
>1.1 S A=7+5  
>1.2 T "ANSWER=",A  
>D 1  
ANSWER=12  
>
```

*SET A to the result of 7+5  
TYPE the contents of A  
DO it*

The SET command:

- creates a variable using the name you chose
- puts your data into it.

The example you just tried is simply a modification of the example from Chapter 1 (1.1 T 7+5, remember?). Let's analyze what happened.

- Step 1.1, the SET (S) command told MUMPS to evaluate the expression '7+5' and create a variable called A to store the result.
- Step 1.2 told MUMPS to type out the contents of variable A.

Of course, we could have used some name other than A if we wanted to -- like Z71 or G9A or ABC or BC or any combination, as long as the first letter is always alphabetic and no more than three characters are used. Another point: if A already had something in it (i.e. it was previously SET), our program would have wiped out the original data and replaced it with 12.

Let's add more to the program.

```
>1.3 S A="7+5"  
>1.4 T "A=",A
```

Write it out and examine it.

```
>W  
1.10 S A=7+5  
1.20 T "ANSWER=",A  
1.30 S A="7+5"  
1.40 T "A=",A
```

*Here's the old program.*

*Change the contents of A.  
TYPE new contents of A. } What we  
just added*

```
>D 1  
ANSWER=12A=7+5  
>
```

*DO it.*

*Notice that the value of A was changed from a numeric variable to a string variable.*

*Also the '=' sign when used with SET means 'assign value of' rather than arithmetic equality.*

Now change 1.4 to improve the format of the output. Add some spaces to the beginning of the literal "A=".

```
>I .4 T " A=",A
```

*Add some spaces. See what happens now!*

```
>D I  
ANSWER=12 A=7+5  
>
```

*Here are the spaces you added.*

Try this one:

First erase the old program.

```
>E  
>I .10 S A=5,B=7,C="A+B="
```

*Use commas to separate the arguments.*

```
>I .20 T "A=",A," B=",B," ",C,A+B
```

*3 spaces*

DO IT.

```
>D I  
A=5 B=7 A+B=12  
>
```

How much can I put in a variable? How do I form a variable name?

**SOME FACTS  
ABOUT VARIABLES**

A variable can contain either a number in the range of  $\pm 21474836.47$  or a string of up to 132 characters.

A variable's name can be from one to three characters. The first character must be alphabetic; the remaining two can be either alphabetic or numeric.

Now that you know how to store data, let's begin our census program. Here's a program to store someone's name.

```
>I .10 S NAME="MILLARD FILMORE"  
>I .20 T "NAME: ",NAM  
  
>D 1  
NAME: MILLARD FILMORE  
>
```

But this isn't much of a census with just one name and no other information. Let's store more information:

```
>I .10 S NAME="MILLARD FILMORE",AGE=173,SEX="M",OCC="U.S. PRESIDENT"  
>I .20 T "NAME:",NAM," AGE:",AGE," SEX:",SEX," OCCUPATION:",OCC  
  
>D 1  
NAME: MILLARD FILMORE AGE: 173 SEX: M OCCUPATION: U.S. PRESIDENT  
>
```

In this example we've defined the variables AGE, SEX, and OCC to contain the additional census data. However, each time new data is entered, we must modify the program:

```
>I .10 S NAME="HERMANN MENSCH",AGE=15,SEX="M",OCC="STUDENT"  
>D 1  
NAME: HERMANN MENSCH AGE: 15 SEX: M OCCUPATION: STUDENT  
>
```

This isn't very practical - MUMPS should do more work. Let's make an automatic census taker. You can write a program so that everyone can enter his own census data from the MUMPS terminal. The READ command lets us do this. READ performs the opposite task of TYPE -- its primary job is to *input* characters from the terminal.



Try this:

```
>E
>I .10 R NAM
>I .20 T " NAME:",NAM


>D 1
```


*One space*

*ERASE the old program.  
READ in a name.  
TYPE out the name.*

*DO it.*

Did anything happen? Can you tell?

Type something, then press  . How about:

*your name* 

What happened? Did it look like this?

```
what you typed
your name NAME: your name
what MUMPS typed
```

When MUMPS ran the program, the first thing it did was to READ from your terminal. (But you couldn't tell this from looking at the terminal.) It simply waited quietly for you to type-in something before doing the rest of the program.

It would be easier to know when a program wanted to READ if some kind of message could be typed.

You could write a program like this:

```
>I .10 T "WHAT IS YOUR NAME?"
>I .20 R NAM
>I .30 T " NAME:",NAM

>D 1
WHAT IS YOUR NAME?ALBERT EINSTEIN NAME:ALBERT EINSTEIN
>
```

*TYPE this*

*There it is!*

Great! Now we know when MUMPS is going to read from the terminal. The first TYPE command isn't really necessary, though. READ can do both jobs.

*Notice!*

*READ also has a secondary job.*

```
>E
>1.10 R "WHAT IS YOUR NAME?",NAM
>1.20 T "  YOUR NAME IS:",NAM
```

```
>D 1
WHAT IS YOUR NAME?JOHN DOE  YOUR NAME IS:JOHN DOE
>
```

Like TYPE, READ can type-out character strings that are enclosed in quotation marks.

So, let's do this. Add an additional argument to step 1.1 to make our program more interactive.

```
>1.10 R "WHAT IS YOUR NAME?",NAM,!
```

*Here's the string to  
be typed out.*

*Here it is!  
Exclamation point*

DO it.

```
>D 1
WHAT IS YOUR NAME?
YOUR NAME IS:
```

*Type your name here*

*Notice -- what was the first thing MUMPS did after you typed in your name? Right! It started printing at the left margin of the line below. This is called a carriage RETURN/LINE FEED operation or CR/LF for short.*

When MUMPS sees an exclamation point character as an argument to a READ or TYPE, it outputs a carriage RETURN/LINE FEED operation. Commas are not needed between adjacent exclamation points (,!!!,).

Try this one:

```
>I .10 R "TYPE YOUR NAME!",!,NAM,!  
>I .20 T "YOUR NAME IS:",!,NAM
```

```
>D !  
TYPE YOUR NAME!  
ELMER FUDD  
YOUR NAME IS:  
ELMER FUDD  
>
```

*This is only an exclamation point -- it's in quotes.*

*These exclamation points mean CR/LF.*

Let's analyze line 1.1 to see what happened. First, MUMPS output TYPE YOUR NAME! . Since the ! following NAME was within the quotation marks, MUMPS printed an exclamation point (!). When MUMPS found the ! enclosed in commas it output a carriage RETURN/LINE FEED. Then it read the name you entered and stored it in variable NAM. Next it found another ! and MUMPS output another carriage RETURN/LINE FEED.

*You figure out the action in step 1.2.*

The READ command can: input data from the terminal to one or more specified variables; output quoted text strings, and output the carriage RETURN/LINE FEED with the ! *Form Control Character*. Each argument is separated from other arguments by commas.

Let's revise our program to include more census data.

```
>I .10 R !!," NAME?",NAM,!, "AGE?",AGE,!, "SEX?",SEX,!, " OCCUPATION?",OCC,!  
>I .20 T !," NAME:",NAM,!, "AGE:",AGE,!, "SEX:",SEX,!, " OCCUPATION:",OCC,!
```

```
>D !  
  
NAME?
```

*This time you type in your own census data.*

NOW your program can automatically store and display census data. But you've only told MUMPS to store one person's census data -- what about the rest of the people in town? A census needs data about a lot of people. How can we store it in memory? The program could be rewritten and run over and over again, each time with different names for the

variables which hold the data. But this could take lots of time and would be a very inefficient use of a computer.

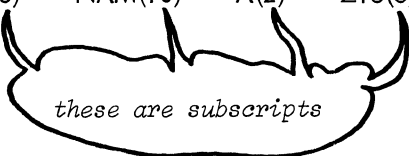
Until now, we have used only *simple variables* with names like:

A B C NAM

Now we want to introduce a new type of variable, called a *subscripted variable*.

Subscripted variables look like:

AGE(3) NAM(10) A(2) Z15(3)



*these are subscripts*

Like simple variables, subscripted variables are also used to store numeric or string data in memory.

Ten subscripted variables

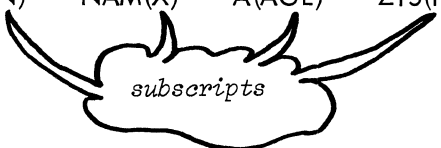
NAM(.01)	data
NAM(.02)	data
NAM(.03)	data
NAM(.04)	data
NAM(.05)	data
NAM(.06)	data
NAM(.07)	data
NAM(.08)	data
NAM(.09)	data
NAM(.10)	data

Ten subscripted variables

AGE(.01)	data
AGE(.02)	data
AGE(.03)	data
AGE(.04)	data
AGE(.05)	data
AGE(.06)	data
AGE(.07)	data
AGE(.08)	data
AGE(.09)	data
AGE(.10)	data

We can also use a variable as a subscript

AGE(N) NAM(X) A(AGE) Z15(P)



*subscripts*

Subscripted variables can be used to create tables of data in memory, called *arrays*. Arrays are useful for storing data in categories.

A subscript can be:

- a constant
- a variable
- an expression

as long as MUMPS can interpret it as a positive number between 0 and 20,975.51.

Here are some of the forms subscripts can take:

AGE((N+1)/4)

NAM(1.01)

SEX(Y)

Back to our census!

The following table contains census information that we would like to have MUMPS store for us. We can use subscripted variables to contain the data in each column.

NAM(N)	AGE(N)	SEX(N)	OCC(N)
HENRY ADAMS	46	M	CARPENTER
BILL SMITH	15	M	STUDENT
ALTHEA BROWN	30	F	CHEMIST
PAUL JOHNSON	22	M	PROGRAMMER
JUDY ZWINK	86	F	GRANDMOTHER
ORPHAN ANNIE	30	F	COMIC STRIP CHARACTER
DADDY WARBUCKS	76	M	WARMONGER
SANDY	15	M	DOG
ZEUS	4000	M	GREEK GOD
IAN MCKENZIE	30	M	IMMUNOLOGIST
BARBARA THOMSON	26	F	SOFTWARE WRITER
MELISSA MERCOURI	12	F	SHOW GIRL
BENJAMIN DOVER	38	M	PROCTOLOGIST
SYLVIA SAWYER	36	F	FARMER
EPHRIAM PRETZELBENDER	30	M	PRETZEL BENDER
KEN GERBER	20	M	NATURALIST
LEROY ABRAMS	30	M	AIRCRAFT MECHANIC
XERXES POLYPHON	30	M	LOUD PLAYER
HELEN TRENT	44	F	BACK STAGE WIFE
MOLLY MALONE	15	F	COCKLE AND MUSSEL VENDOR
CYNTHIA SMECK	27	F	LUMBERJACK
ALICE JOHNSON	34	F	ARTIST
MARK ALTMAN	20	M	TREASURER
NATE LISKOV	34	M	MATHEMATICIAN
NANCY PENN	20	F	RADICAL
JOHN FAVOR	38	M	SAW MILL OPERATOR
MINNIE PEASLEE	42	F	TOWN CLERK
KEN MASER	27	M	MINING ENGINEER
EVELYN JORDAN	30	F	PHOTOGRAPHER
HIRAM WALKER	60	M	WHISKEY TASTER

Here is a program that stores the census data (above) in memory. Type it in.

```

>E
>1 .10 S N=.01
>1 .20 R !,"NAME:",NAM(N),!,"AGE:",AGE(N),!,"SEX:",SEX(N),!
>1 .25 R "OCCUPATION:",OCC(N),!
>1 .30 I !,NAM(N)," ",AGE(N)," ",SEX(N)," ",OCC(N),!
>1 .40 S N=N+.01

```

Before running the program, let's analyze it. In step 1.1 we define simple variable 'N' and give it an initial value of .01. Since we will be storing census data on many people, we'll use 'N' as a counter.

Step 1.2 types out messages requesting data (name, age, etc.) and reads the data into subscripted variables (NAM(N), AGE(N), etc.), using the current value of 'N'. Step 1.3

echoes the data just input to allow verification. Step 1.4 increases the counter 'N' by one, in preparation for input of the next data record.

OK, run the program. Type:

>D 1

NAME: HENRY ADAMS  
AGE: 46  
SEX: M  
OCCUPATION: CARPENTER

*Type-in a name from the table.  
and age  
and sex  
and occupation*

HENRY ADAMS 46 M CARPENTER

*Here's what's stored in memory.*

>

Good, you've entered and stored the census data for one person. But what about the other people. The program, like others in previous examples, stopped when it ran out of steps. If we start it manually by typing D 1, that won't help either. The data we put in the first time around will be wiped out because the counter 'N' will be reinitialized to a .01. Certainly there are other manual solutions to this problem, but the point is to make MUMPS do the work -- automate the job.

Here's how. Add a new step at the end of the program:

>1.5 G 1.2

Step 1.5 says "GOTO step 1.2 and continue from there"; 'G' is the GOTO command. Every time MUMPS finds a GOTO in a program, it 'goes to' the specified step and continues.

GOTO unconditionally transfers program control to the specified step number. Needless to say, the step numbers must be legal and part of the program.

Now, write out the program.

```
>W
1.10 S N=.01
1.20 R !,"NAME:",NAM(N),!,"AGE:",AGE(N),!,"SEX:",SEX(N),!
1.25 R "OCCUPATION:",OCC(N),!
1.30 T !,NAM(N)," ",AGE(N)," ",SEX(N)," ",OCC(N),!
1.40 S N=N+.01
1.50 G 1,2
```

Examine the program. Follow it step-by-step. Think it through. OK! Before running the program, let's make one final change. We're going to change step 1.3 so it will type out the value of N for each data item. This will help us to know where the data is stored in the array.

Here's the revised step:

```
>1.30 T !,N," ",NAM(N)," ",AGE(N)," ",SEX(N)," ",OCC(N),!
```

*Here's the change.*

Type it in.

Good! Run the program.

Type:

```
>D 1
NAME: HENRY ADAMS
AGE: 46
SEX: M
OCCUPATION: CARPENTER

0.01 HENRY ADAM 46 M CARPENTER

NAME:
AGE:
SEX:
OCCUPATION:
.
.
.
```

*Here we go. You enter the data. Use the census information in the table on page 2-11.*

*You put it in.*



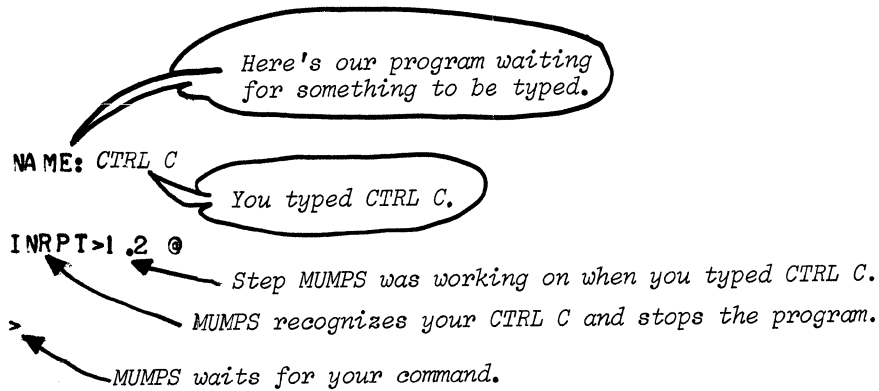
Once you've entered census data for five or six people, stop the program. Let's go on to something else. What? You don't know how to stop? That's right! Our other programs always stopped because they ran out of steps. But in this program we asked you to put a GOTO in step 1.5. Now the program never runs out of steps. Our program is looping endlessly through steps 1.2 to 1.5, then back to 1.2. Programmers often call this an "infinite loop".

```
1.10 S N=.01  
1.20 R !,"NAME:",...  
1.25 R "OCCUPATION:",0C ...  
1.30 T !,NAM(N) ...  
1.40 S N=N+.01  
1.50 G 1.2
```

*There's the loop.*

Here's how you stop a program that's running this way. Remember in Chapter 1 we told you to use CTRL C to tell MUMPS that you want to use a terminal? Well, the same thing applies here. This time, you need to tell MUMPS that you want to take control of the terminal and stop the program -- so type CTRL C.

*Hold down the CTRL key and type a C.*



CTRL C can be used any time you want to stop a program.

## CHAPTER 3

### MAKING DECISIONS

Up till now, the programs you've written could input and store data (READ and SET), perform arithmetic operations (+ - \* / ) on it, and type it out (TYPE). One thing your programs couldn't do was make decisions.

Decisions like:

- Is A greater than B?
- Did someone type-in the wrong number or string?
- Are there too many characters in a name?
- What is the relationship of one arithmetic expression to another?
- Is it time to stop a program?

Almost all programs need to make decisions like these.

The IF command is one way that MUMPS lets you make decisions in your programs.

Try this:

```
>E
>I.10 R "A=",A," B=",B,!
>I.20 I A=B I "EQUAL",! G I.!
>I.30 T "UNEQUAL",!
>I.40 G I.!
```

*First ERASE any old programs.*

*Here's the IF command.*

```
>D I
A=5 B=4
UNEQUAL
A=DOG B=DOG
EQUAL
A=
```

*Your turn. Type anything you want to.*

Have you had enough? OK. Then stop the program.

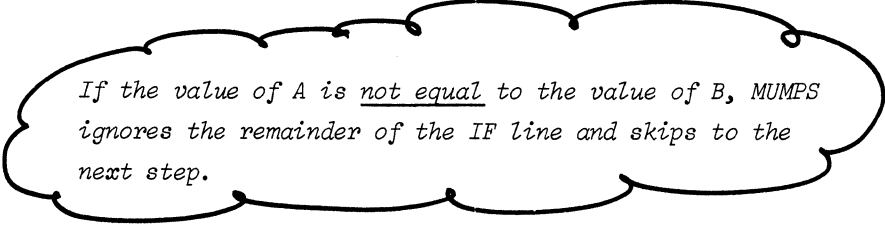
*Type CTRL C.  
Remember?*

Step 1.2

```
1.20 I A=B T "EQUAL",! G 1.1
```

tells MUMPS:

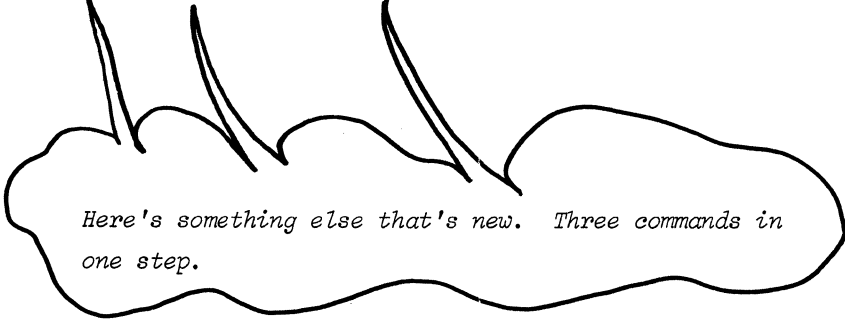
IF THE VALUE OF A IS EQUAL TO THE VALUE OF B, THEN TYPE "EQUAL", DO A CR, AND GO TO STEP 1.1.



*If the value of A is not equal to the value of B, MUMPS ignores the remainder of the IF line and skips to the next step.*

Look at the line again.

```
1.20 I A=B T "EQUAL",! G 1.1
```



*Here's something else that's new. Three commands in one step.*

Actually, you can put as many commands in a step as you wish -- up to 132 characters. Just separate each command from the next with a single space.

Now, let's change the program:

```
>1.30 I A>B T A-B,! G 1.1
>1.40 T "A LESS THAN B",!
```

WRITE it out.

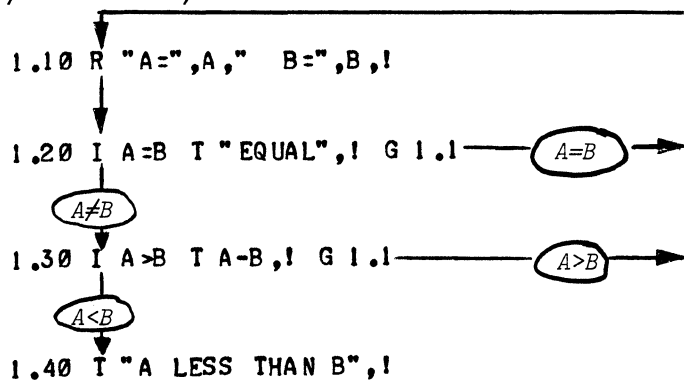
```
>W
1.10 R "A=",A," B=",B,!
1.20 I A=B T "EQUAL",! G 1.1
1.30 I A>B T A-B,! G 1.1
1.40 T "A LESS THAN B",!
```

```
>D 1
A=
```

*Fill in the answers yourself. Can you figure out what answers will make the program stop?*

If A is equal to B, the message "EQUAL" is typed and new values are requested. If A is greater than B, the difference (A - B) is typed out and new values are again requested. Otherwise, A must be less than B. Since there are no more steps left after 1.4, the program stops when A is less than B.

To say it another way:



*This is a relational expression.*

1.20 I A=B T "EQ ... → Follow this path if relation is TRUE.

Follow this path if relation is FALSE.

*This is a relational operator.*

Another view of the IF command.

General Form: IF *TRUE relation* do next command on the line, otherwise do next step.

Example: I ANS= "YES" T NAM(N)

If variable ANS contains the string "YES", the contents of NAM(N) will be typed.

Relational expressions are simply expressions which contain *relational operators*. The relational expression used with IF can be an arithmetic relation between two arithmetic expressions. Here are the arithmetic operators that can be used with IF.

Relation	Symbol
Equal to	=
Less than	<
Greater than	>
Less than or equal to	<= or =<
Greater than or equal to	>= or =>
Not equal to	<> or ><

Here's another example:

```
>E
>1 .10 R "A=",A," B=",B,!
>1 .20 I A<100,B>0 T A*B,! G 1.1
>1 .30 T "OUT OF RANGE",!
>1 .40 G 1.1
```

```
>D 1
A=75 B=1
75
A=54 B=-12
OUT OF RANGE
A=
```

*Erase the old program.*

*IF A is less than 100 and B is greater than 0, TYPE A\*B. Notice that IF can include more than one relation. Just separate each expression with a comma.*

In this example, IF has two arguments:

```
A < 100
and
B > 0
```

If both arguments are TRUE, the TYPE command is executed and MUMPS skips to 1.1. If either argument is FALSE, the TYPE command is ignored and MUMPS skips to 1.3, performs it, then goes to 1.1 as commanded in line 1.4. Try the example again with other values.

Now write some examples of your own, using IF. Try using variables, constants, literals, and arithmetic operators in various combinations.

In Chapter 2 we showed you a program to collect and store census data. The program ran very well except that you had to type CTRL C to stop it. The problem with using CTRL C to stop a program is that it may not stop it at a logical point -- the program might have stored a name, but not age, sex, or occupation. That could corrupt our data.

Here's the old census input program:

```
1.10 S N=.01
1.20 R !,"NAME:",NAM(N),!,"AGE:",AGE(N),!,"SEX:",SEX(N),!
1.25 R "OCCUPATION:",OCC(N),!
1.30 I !,N," ",NAM(N)," ",AGE(N)," ",SEX(N)," ",OCC(N),!
1.40 S N=N+.01
1.50 G 1.2
```

Let's change it so that it can be told when to stop. We'll use the IF command. Replace step 1.5:

```
>1.50 R !,"MORE DATA (Y OR N)?",ANS,! I ANS="Y" G 1.2
```

This step tells MUMPS to ask if more data is to come. (You should answer with a Y for yes or N for no.) If your answer is a Y, MUMPS starts the program over at step 1.2. Any other answer causes the program to stop since there are no more steps after step 1.5.

Run the program.

>D 1

NAME: \_\_\_\_\_  
AGE: \_\_\_\_\_  
SEX: \_\_\_\_\_  
OCCUPATION: \_\_\_\_\_  
:  
:

}

*You provide the  
data this time.*

MORE DATA (Y OR N)?

*Do you want to stop, or go on?  
It's up to you!*

Now we can tell the program to stop itself rather than asking MUMPS to do it for us.

Let's think a little about what the census input program is doing for us.

- It reads in data.
- It stores the data.
- It types out the data so we'll know what's been stored.
- It stops on command.

What good does the stored data do? How is it useful? It's not useful by itself, just sitting there in the computer memory. The purpose of a census is to gather lots of up-to-date information about lots of people so we can tell what the characteristics of a population are.

- How many carpenters between ages 32 and 39 are women?
- What is the most common occupation?
- What is the average age?

The reason should be obvious now. Since MUMPS can do things faster and more accurately than people can, it is natural to have it update, retrieve, and report our census information. The Input program is just one of the programs needed to handle our census operation -- it stores the data that other programs will use.

Remember what our *data base* looks like?

If you've forgotten, look back in Chapter 2. Try to imagine how a program to count the number of people in the census might work. All that is required is to count the number of names in the data base. How? Simply start at the beginning, look at each entry, and if there's a name there, count it. If there's no name or something which doesn't resemble a name, don't count it -- this is the end. And that's our problem -- how to know where the end of the data file is.

If the program that did the counting looked like this:

```
>1 .10 S N=.01
>1 .20 I NAM(N)="*" G 1.4
>1 .30 S N=N+.01 G 1.2
>1 .40 T "TOTAL=",N,!

```

it would find the end of the data if the last entry were an asterisk, for example. The idea is to make the last entry a "dummy data record" with something in it which is not normally considered data -- like an asterisk.

Here's our version of the data input program. It not only stops on command, but also writes a "dummy record" as well. Read it through carefully.



```

1.10 T !,"CENSUS DATA INPUT PROG",!!," TO STOP, TYPE A CR"
1.15 T "IN RESPONSE TO 'NAME:' REQUEST",!
1.16 S N=.01
1.20 R !," NAME:",NAM(N),!

```

*This means null string  
or (no characters)*

*Here's our asterisk*

*This is the QUIT command*

```

1.25 I NAM(N)=" " S NAM(N)="*" Q
1.28 R "AGE:",AGE(N),!,"SEX:",SEX(N),!,"OCCUPATION:",OCC(N),!
1.30 T N," ",NAM(N)," ",AGE(N)," ",SEX(N)," ",OCC(N),!
1.40 S N=N+.01 G 1.2

```

Type it in, but ERASE first.

```

>E
>1.1 T !,"CENSUS...
.
.
.

```

Now run it.

```

>D 1
CENSUS DATA INPUT PROG
TO STOP, TYPE A CR IN RESPONSE TO 'NAME:' REQUEST
NAME:

```

*Now go back to the table of census data in Chapter 2 and enter all the data. When you're finished, stop the program. Just type a CR only in response to the 'NAME' request.*

Step 1.25 does a lot of work.

```
1.25 I NAM(N)="" S NAM(N)="*" Q
```

Let's analyze it. First, remember that the way we stopped in earlier versions of this program was to look for a "Y" or "N" response. "Y" and "N" are character strings. What if the program looks for a string of no characters -- this kind of string "" . This "" is a null string. There aren't any characters in it. When MUMPS does a READ and you type a CR only, a null string is input. *Step 1.25 tells MUMPS if a null string is read into NAM(N), that's the signal to stop. But before stopping, put an asterisk into NAM(N); then QUIT.*

QUIT is a new command.

When MUMPS sees a QUIT, it stops the program and gives terminal control back to you.

Our data base looks like this now:

HENRY ADAMS	46	M	CARPENTER
BILL SMITH	15	M	STUDENT
ALTHEA BROWN	30	F	CHEMIST
PAUL JOHNSON	22	M	PROGRAMMER
.	.	.	.
.	.	.	.
.	.	.	.
KEN MASER	27	M	MINING ENGINEER
EVELYN JORDAN	30	F	PHOTOGRAPHER
HIRAM WALKER	60	M	WHISKEY TASTER

*\*  
Last data entry*

*Dummy record*



## CHAPTER 4

# CONSERVATION OF ENERGY

### TIRED HANDS

Up till now you've been doing a lot of typing and retyping of programs. Some of them have long lines which aren't very different from one another. That's a tiring and often boring job. A lot of this drudgery can be avoided.

MUMPS gives you a program storage area in memory all for yourself. Each time you type-in a program step or create a new variable, more of your storage area gets used up. Since MUMPS is a time-sharing system, you have a limited amount of memory. There are other users like yourself, so memory is divided into *partitions* -- one partition for each of you. When you DO a program, MUMPS looks for it in your partition. It thinks that all the steps in there are part of the same program. So, when you write a new program, if you don't want the old program to be a part of it, you have to ERASE. Then if you want to use the old program again, it must be retyped. Also, when you sign-off (log-out) at a MUMPS terminal, any program in your partition is wiped out.

"What's all this leading up to?" you ask. Just this:

Memory is for temporary storage of programs. So that you don't have to continually type-in the programs that you want to run, MUMPS lets you save them permanently in disk storage.

In Chapter 1 we told you about logging-in (sign-on) and we also told you what a UCI is. Well, MUMPS gives everyone who has a UCI a storage area on the disk. You can store an almost unlimited number of programs there. Give your tired hands a rest.

The FILE command stores programs on the disk. The LOAD command brings them back.

Is the Census Data Input program still in memory? Try to WRITE it out.

Type:

>W

```
1.10 T !,"CENSUS DATA INPUT PROGRAM",!!,"TO STOP, TYPE A CR "  
1.15 T "IN RESPONSE TO 'NAME:' REQUEST",!  
1.16 S N=.01  
1.20 R !," NAME:",NAM(N),!  
1.25 I NAM(N)=" " S NAM(N)="*" Q  
1.28 R "AGE:",AGE(N),!,"SEX:",SEX(N),!," OCCUPATION:",OCC(N),!  
1.30 T N," ",NAM(N)," ",AGE(N)," ",SEX(N)," ",OCC(N),!  
1.40 S N=N+.01 G 1.2
```

>

If it's not there, type it now.

Before we can FILE a program, we must pick a name for it. Program names follow the same rules as variable names (except for subscripts, of course). How about calling the program INP?

Type:

```
>F INP  
>
```

*one space*

*Don't forget*

RE-TURN

Now the Census Data Input program is in your storage area on the disk under the name INP. The original copy is still here too -- in your partition. WRITE it out and see.

```
>W  
1.10 T !,"CENSUS DATA INPUT PR...  
.  
.  
.
```


You don't need this one -- ERASE it.

>E

Is it gone? Try to WRITE it and see for yourself.

>W

>



*No program!  
It's ERASEd.*

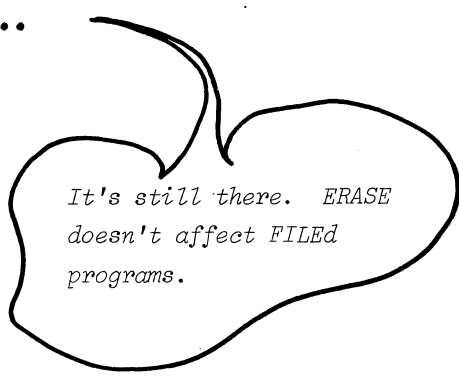
Now, let's try to get the program back from disk storage. Let's LOAD it.

>L INP

>W

1.10 T !,"CENSUS DATA INPUT PR...

.  
.  
.



*It's still there. ERASE  
doesn't affect FILEd  
programs.*

OK. ERASE again.

>E

>

Here's a new twist. Try this.

LOAD it again.

>L INP

>

Then type:

```
>F IN1
```

```
>
```

Do you know what you did? Right! You just FILEd the INP program under another name, IN1.

Now there are two copies of the same program with different names. You can make as many copies as you want.

Like this:

```
>F IN2,IN3,IN4,ZZZ
```

```
>
```

Now there are six copies of the original INP program. Try LOADING them and WRITing them out if you wish.

We don't need all these copies. Let's ERASE one. How about ZZZ? First empty your partition.

Type:

```
>E
```

```
>
```

then type:

```
>F ZZZ
```

```
>
```

It's ERASEd! If you don't believe it, try to LOAD ZZZ.

```
>L ZZZ
NOPGM>0 @
>
```

*See! It's not there.*

ERASE the rest except for INP.

```
>F IN2,IN3,IN4
>
```

*All ERASEd.
Check for yourself.*

```
>L IN2
NOPGM>0 @
>L IN3
NOPGM>0 @
>L IN4
NOPGM>0 @
>
```

To erase a FILEd program:

1. ERASE your partition.
2. Do a FILE using the name of the program.

## CHANGING TIMES

People are always changing - people change jobs, people are born, people change names, people get older. All these changes keep census takers busy updating their data.

What do we mean by update?

- Adding new data
- Modifying or correcting existing data

Now that we've got a program that creates our initial census data base, how can the data be kept up to date? We have to be able to: *add* new data and *modify* existing data.



First, we need a program to add new census data to the existing census data that MUMPS stored for us.

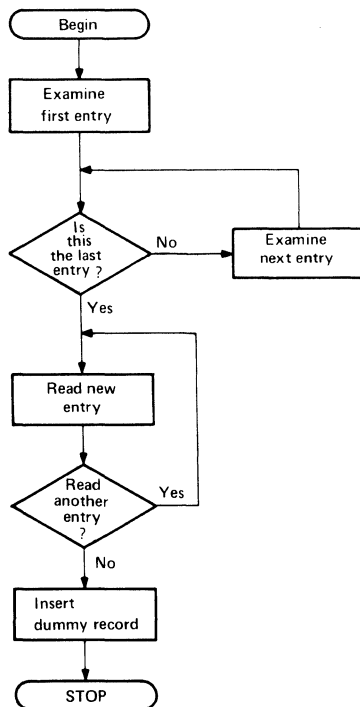
Think about what a program has to do to keep the census up to date.

Here's what we think:

It must:

- Search through the data file until the end is found (remember we made a 'dummy record' in the last NAM(N) entry?).
- Delete the 'dummy record'.
- Read in the new name, age, sex, and occupation.
- See if there is another entry to be added. If there is, add it.
- If there isn't, write a new dummy record and stop.

Here's a diagram of what our program does:



11-1977

You write your own version of the program -- we'll write ours below:

```
>2.10 S N=.01
>2.20 I NAM(N)="*" G 2.4
>2.30 S N=N+.01 G 2.2
>2.40 R !," NAME:",NAM(N) I NAM(N)=" S NAM(N)="*" Q
>2.50 R !," AGE:",AGE(N),!," SEX:",SEX(N),!," OCCUPATION:",OCC(N),!
>2.60 S N=N+.01 G 2.4
>
```

How does yours compare? Run your program. Does it work? Can you fix it if it doesn't? Try our program now, but first FILE yours so you can work on it later. Pick a name for your program. Type F \_\_\_\_\_ Now ERASE and type in our program.



Now run our program.

Type:

```
>D 2 DO all the steps beginning with 2.
```

```
NAME:
```

Did it run?

If you got an

```
UNDEF>2.20 @
>
```

error, chances are that your data base was destroyed, most likely because you signed-off your terminal by typing HALT (H). Programs and variables in your partition are destroyed when you log-out.

We'll tell you about permanent data storage in the next chapter. But for now you'll have to

1. FILE theADD program  
Type:  
**>F ADD**
2. LOAD the INP program.  
Run it and enter at least one person's data.
3. After stopping INP, LOAD ADD.  
Now we're ready to continue.

If the program ran OK, so much the better. But do you know if the census data you added is really there?

First, stop the program. Type a null string (CR only) when the program types the NAME: request.

Now, let's look and see if the program did its job.

Type:

```
>I N
0.31
>
```

*This is the subscript value of the last entry -- the dummy record. The last data entry is N-.01.*

Look at N-.01.

Type:

```
>I NAM(N-.01)
HIRAM WALKER
>
```

*Notice we didn't put the TYPE command in a program step. This tells MUMPS to execute the command immediately or directly.*

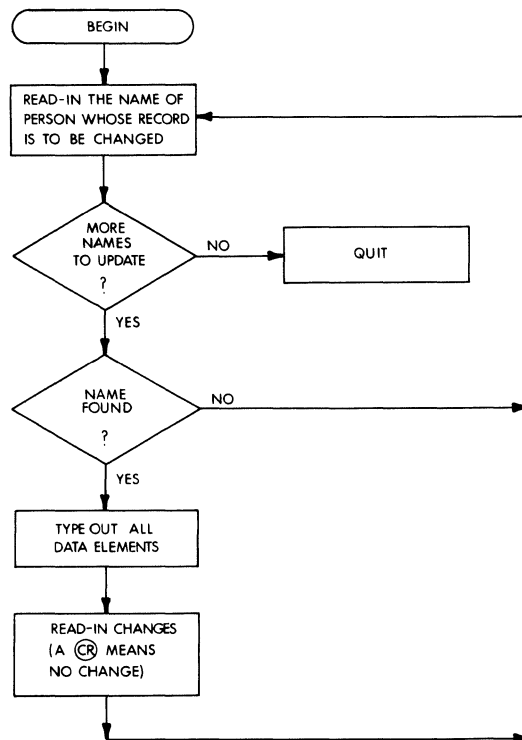
Is this the last name you entered? Good. The program works. If it isn't, make sure there isn't an undetected typing error in the program. WRITE it out and see.

Look at the remaining data elements in the entry:

```
>T AGE(N-.01),!,SEX(N-.01),!,OCC(N-.01)
S0                               Age
M                               Sex
WHISKEY TASTER                   Occupation
>
```

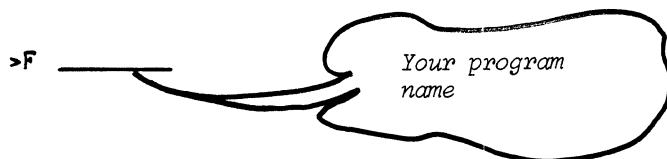
Is the data OK? Does the ADD program work?

Good! Let's go on. The remaining requirement for a data update task is to modify an existing entry. This lets you keep current census data up to date. Draw a flow diagram of the way you think this program should work -- we'll do ours below.



Write the program the way you diagrammed it. Use step numbers beginning with 3.1. Then FILE it. You pick the name, but don't use 'MOD' - we're going to use that.

Type:



Here's the way we wrote the program.

Type it in, but first ERASE.

```
>E
>3.10 S N=.01 R !!!,"UPDATE DATA FOR:",!, "NAME:",NAM
>3.20 I NAM="" Q
>3.22 I NAM(N)=NAM G 3.4
>3.30 I NAM(N)="*" I !,"NOT FOUND", G 3.1
>3.32 S N=N+.01 G 3.22
>3.40 I !,NAM(N)," ",AGE(N)," ",SEX(N)," " OCC(N),!
>3.50 R !,"CHANGES TO BE MADE",!, "NAME:",NAM I NAM="" G 3.6
>3.52 S NAM(N)=NAM
>3.60 R !,"AGE:",AGE I AGE="" G 3.7
>3.62 S AGE(N)=AGE
>3.70 R !,"SEX:",SEX I SEX="" G 3.8
>3.72 S SEX(N)=SEX
>3.80 R !,"OCCUPATION:",OCC I OCC="" G 3.1
>3.82 S OCC(N)=OCC G 3.1
>
```

Now FILE it as MOD for MODify.

```
>F MOD
>
```


Here's how MOD works:

- Step 3.1 reads-in the name of the person whose census data is to be changed.
- Step 3.2 stops the program if a null string is entered.
- Step 3.22 compares the name read-in from the terminal with an entry in the data base. If the names are the same, processing continues at step 3.4.
- Step 3.2 IF the names aren't the same, this step checks to see if the dummy record at the end of the data was reached. If it was, the message "NOT FOUND" is typed and the program begins again.
- Step 3.32 If the end of the data was not reached, increment the counter (N) by .01 and examine the next data base entry.
- Step 3.4 Once a match is found, the entire census data for that person is output so you can verify the data.
- Steps 3.5 through 3.82 read-in the replacement data. If you enter a null string in response to the messages "NAME:", "AGE:", etc., the old data entry is retained. When all responses have been made, the program begins again.

Now run the program. Try yours first.

Type:

```
>L  
>D 3
```



*Your program name*

Change several entries in the data base. Then stop the program and use TYPE to examine the changed data as we did when we ran the ADD program. Does your program work OK? If not, try to fix it. Then run it again, check it out, and FILE it again so you'll have saved the latest copy.

Our turn. Run our program.

>L MOD

>D 3

UPDATE DATA FOR:  
NAME: ANDREW JACKSON  
NOT FOUND

UPDATE DATA FOR:  
NAME: CYNTHIA SMECK  
CYNTHIA SMECK 27 F LUMBERJACK

CHANGES TO BE MADE  
NAME: \_\_\_\_\_  
AGE: 34  
SEX: \_\_\_\_\_  
OCCUPATION: CAB DRIVER

*null string entered --*

*no change*

UPDATE DATA FOR:  
NAME: \_\_\_\_\_

*null string entered*

*here stops the program*

*Your turn.*

## A NEW TWIST

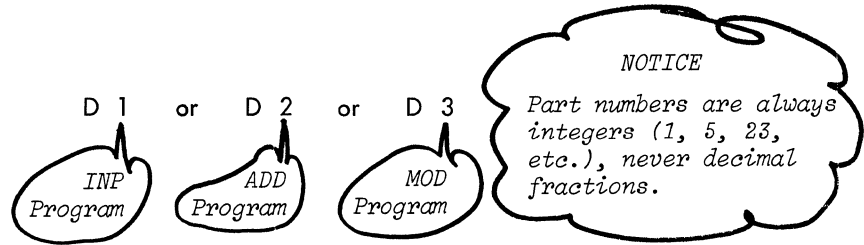
If you've been observant, you may have noticed something about the step numbers used in the INP, ADD, and MOD programs. The integer *parts* of the step numbers changed from 1 to 2 to 3. That is, INP used numbers like 1.1, 1.2, etc., while ADD used numbers like 2.1, 2.2, etc., and MOD used numbers like 3.1, 3.2, etc. The significance of these differences is that MUMPS treats program steps that have a common integer base as a unit. All MUMPS programs can be divided up into *parts* using this numbering scheme.

Numbers like 1.2      1.37      1.90      etc.



are in part 1, while part 5 contains numbers like 5.01, 5.53, etc.

You may also have noticed that when we told you to DO the INP, ADD, and MOD programs you typed:



When we run a program that has more than one part, the program will stop when it reaches the end of the first part. MUMPS doesn't do the other parts unless we tell it to DO them or to GOTO them.

DO is one way you can tell MUMPS to *execute* the steps in another program part. When all the steps are DOne, MUMPS returns to the command immediately following the DO. A DO can also be used to DO a specific step.

<code>&gt;D 1.37</code>	<i>DO step 1.37</i>
<code>2.1 D 4,1,3.5</code>	<i>DO part 4, part 1, and step 3.5</i>
<code>&gt;D 6.3,8,2.1</code>	<i>DO step 6.3, part 8, and step 2.1</i>

A QUIT can prematurely terminate execution of the steps in the range of a DO, causing MUMPS to return to the command immediately following the DO.

<code>1.1 D 2,5 I "DONE",!</code>	<i>DO parts 2 and 5, then type "DONE" and RETURN</i>
<code>2.1 I A=2.5 Q</code>	<i>IF A=2.5 QUIT - don't DO the rest of part 2 - go back and do the rest of step 1.1.</i>

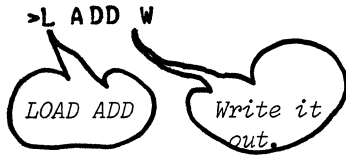
GOTO is also used when you want to execute steps in another part. Unlike DO, control does not automatically return to the command following the GOTO.

Back to our census programs.

Let's be sophisticated and combine the ADD and MOD programs into one program. Since each uses different part numbers, there'll be no conflicts.



So, LOAD the ADD program and WRITE it out.



Now load the MOD program.

```
>L MOD
>
```

Now let's *merge* the two programs. Just type in the steps contained in ADD. When you've finished, FILE this new program as UPD.

Type:

```
F UPD
>
```

To do a census update, simply LOAD UPD and DO either part 2, if the data base is to receive additions, or part 3, if modifications are desired.

There's one more thing that can be done to make the UPD program more automatic. Let's create a part 1 in the program to decide which part (2 or 3) to DO. Make sure that UPD is LOAded, then type:

```
>1.10 I !,"CENSUS DATA UPDATE PROGRAM",!
>1.20 I "OPTIONS: A=ADD DATA M=MODIFY DATA",!!
>1.30 R "OPTION*",ANS I ANS="A" D 2 DO part 2
>1.40 I ANS="M" D 3
>1.50 I ANS="" I !,"GOOD BYE",! Q
>1.60 G 1.2 DO part 3
>
```

Write out the entire program. Read it through -- make sure there aren't any errors.

```

>W
1.10 T !,"CENSUS DATA UPDATE PROGRAM",!
1.20 T "OPTIONS: A=ADD DATA M=MODIFY DATA",!!
1.30 R "OPTION* ",ANS I ANS="A" D 2
1.40 IF ANS="M" D 3
1.50 I ANS="" T !,"GOOD BYE",! Q
1.60 G 1.2

2.10 S N=.01
2.20 I NAM(N)="*" G 2.4
2.30 S N=N+.01 G 2.2
2.40 R !,"NAME:",NAM(N) I NAM(N)=" " S NAM(N)="*" Q
2.50 R !,"AGE:",AGE(N),!,"SEX:",SEX(N),!,"OCCUPATION:",OCC(N),!
2.60 S N=N+.01 G 2.4

3.10 S N=.01 R !!!,"UPDATE DATA FOR:",!,"NAME:",NAM
3.20 I NAM="" Q
3.22 I NAM(N)=NAM G 3.4
3.30 I NAM(N)="*" T !,"NOT FOUND", G 3.1
3.32 S N=N+.01 G 3.22
3.40 T !,NAM(N)," ",AGE(N)," ",SEX(N)," ",OCC(N),!
3.50 R !,"CHANGES TO BE MADE",!,"NAME:",NAM I NAM=""# G 3.6
3.52 S NAM(N)=NAM
3.60 R !,"AGE:",AGE I AGE="" G 3.7
3.62 S AGE(N)=AGE
3.70 R !,"SEX:",SEX I SEX="" G 3.8
3.72 S SEX(N)=SEX
3.80 R !,"OCCUPATION:",OCC I OCC="" G 3.1
3.82 S OCC(N)=OCC G 3.1

>

```

Now FILE it; then run it. Experiment!

```

>F UPD D 1

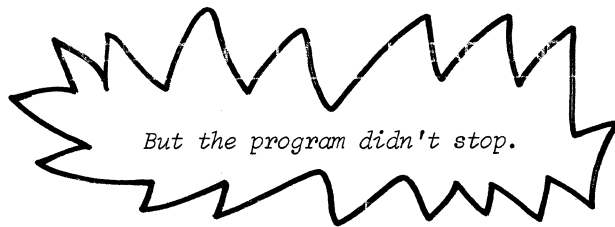
CENSUS DATA UPDATE PROGRAM
OPTIONS: A=ADD DATA M=MODIFY DATA

OPTION*A _____ Select A option.
NAME: _____
AGE: _____ } You fill in the blanks.
SEX: _____
OCCUPATION: _____

NAME: _____ Type a null string.
OPTIONS: A=ADD DATA M=MODIFY DATA

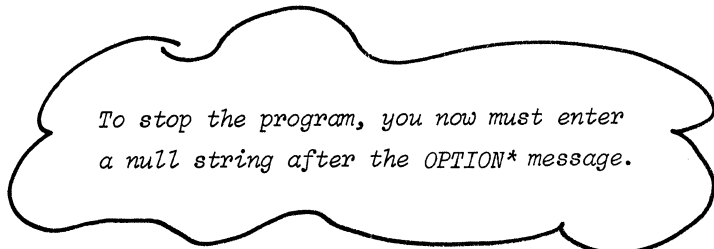
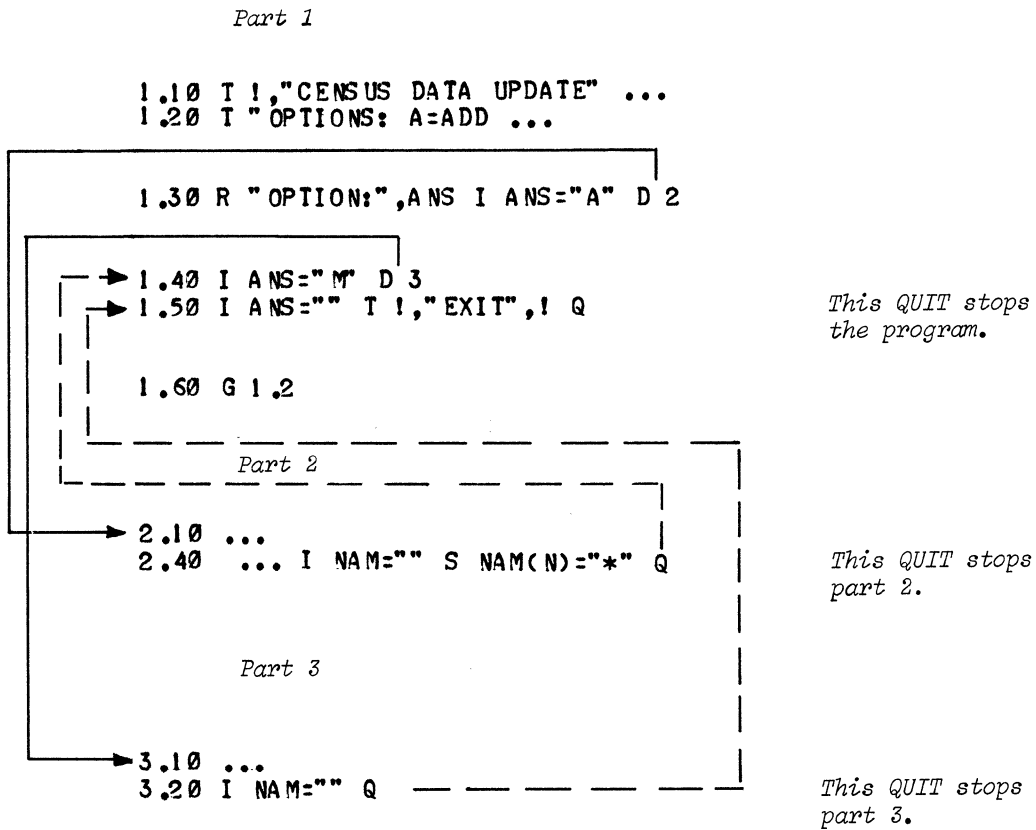
OPTION*

```



It went back to the command following the DO (in part 1). Since there was another step following that DO, MUMPS didn't stop the program -- *it did the step, instead.* Remember what we told you about DO a few pages back?

If you're still confused, the diagram below may help. It shows the lines of control created by the DO and QUIT commands in the three parts of the program.



## CHAPTER 5

# GETTING IT TOGETHER

Let's evaluate our two census application programs. Here's what they do:

They

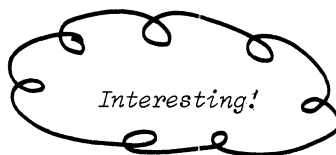
- create the data base
- update the data base by allowing addition of new data and modification of existing data

That's fine. Both these programs satisfy specific functional requirements of our census application. Now, let's look at them from another point of view. How *well* do they do their jobs? WRITE them out. Flowchart them! Examine them closely. Compare them, instruction for instruction. Are there similarities, redundancies, duplications? These can waste valuable space in main memory and make the programs run longer.

So what? You say, "The computer will still do the work." Yes, but remember this will ultimately cause the people who use your programs to waste their time waiting for your slow programs to do their jobs. When this happens, your computer is less valuable to its application.

Do you see ways to improve the programs? Can you make them run faster? Can you eliminate wasted space? We can! There are some very interesting similarities between the INP program and the ADD *subroutine* of the UPD program.

- They both request the same kind of data.
- They both put the data into the same data base.
- They can even be stopped the same way.



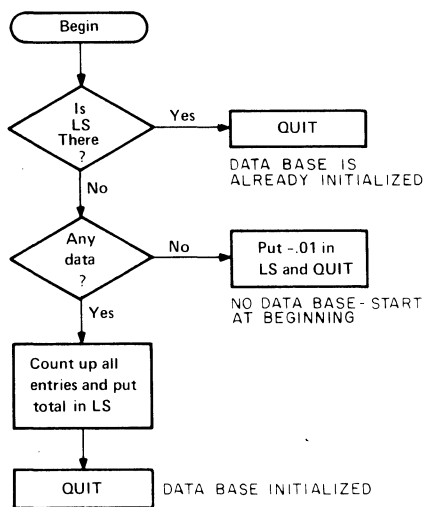
The only significant difference between them is that INP creates the data base when none exists, while ADD simply enlarges an existing data base. In truth, they both *add* to the data base. It's simply a matter of where they begin. INP assumes the data base begins

at subscript .01 (i.e., NAM(0.01), AGE(0.01), etc.). ADD assumes the data base begins at the dummy record (i.e., the asterisk (\*), remember?). We think that the function performed by INP is just a special case of the function performed by ADD. If we always knew where the end of our data base was, INP could be eliminated. If we knew how large the data base was, in terms of its highest subscript, the dummy record containing the asterisk could also be eliminated. This would speed up the operation of ADD, since it wouldn't have to search each entry in the data base to find the \* at the end whenever someone wanted to add a new data entry.

What we need is a program that determines (a) whether a data base exists at all, and (b) if it does exist, where it ends. Why not write a program that determines the highest subscript in the data base and stores this value in some variable? The fact that no data base exists could be signified by storing some nonsubscript value like -0.01 in the variable. Then the UPD program and any others we might want to write could simply examine this variable rather than scan the whole data base.

Could you write this *data base initialization program* using your knowledge of MUMPS? Think about it! What are the steps to be done? Let's flowchart the program first - like this:

*We'll use a variable called LS to store the value of the last subscript.*



11-1979

Here's how we think the program should work (you follow on the flowchart).

First, we'll say that LS normally contains the value of the last (highest) subscript in the data base. If there is no data base, LS will contain a -.01. Now, in either case, as long as LS exists, our programs will know the status of the data base and can act accordingly.

Here's how the program operates:

- If LS is defined, we'll assume it contains the correct data -- so QUIT.
- Otherwise, we'll check to see if there is any data base. If there isn't, put -.01 in LS and QUIT.
- Otherwise, count up all entries in the data base, put the total (highest subscript) in LS, then QUIT.

Can you figure out what commands to use to write the program? Unless you know more than we've told you about MUMPS, you probably can't.

We're going to have to tell you more about MUMPS. But first, are there any commands you *do* know that could be used here? If you said IF -- you're right. We'll probably need IF in several places to make those decisions shown in the flowchart. Let's take the first one. "Is LS there?"

Remember what the general form of IF is?

IF *TRUE condition* do next command;  
otherwise, do the next step.

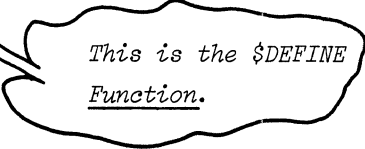
The TRUE condition in this case is: LS *is* there.

*How can we find out whether LS is there?*

MUMPS has a special *function* that can tell us about the existence of LS, or any other variable, for that matter.

Type this:

```
>S LS=0           Create LS.
>T $D(LS)        If LS is DEFINED, type a 1; if not,
1                type a 0.
>
```



*This is the \$DEFINE  
Function.*

Let's analyze what happened:

First, we defined (or created) LS by setting it to a value -- 0 in this case.  
Then we told MUMPS to TYPE the result of the expression \$D(LS).

This expression:

**\$D(LS)**

will produce a result between 1 and 7 if LS is defined,

or

it will produce a 0 result if LS is not defined. *Right now it's not important for you to know what the significance of the numbers 1 to 7 is. The important concept is that a TRUE result is nonzero. In this case, MUMPS typed a 1, which means that LS is defined.*

A *function* is a set of procedures *built into* MUMPS to perform a specific task. Unlike commands, functions operate on expressions and expression elements (variables), and functions can themselves be elements of other expressions.

What if LS is not defined? Let's eliminate LS. Delete it! Destroy it! KILL it!

The KILL command is used to delete MUMPS<sup>1</sup> variables.  
General form: K v<sub>1</sub>, v<sub>2</sub>, v<sub>3</sub>, etc.  
v = name of variable to be KILLED

WARNING  
*IF YOU DON'T SPECIFY ANY VARIABLE NAME(S),  
ALL YOUR VARIABLES WILL BE KILLED.*

Now, KILL LS, then use \$D to see if it was KILLED.

Type:

```
>K LS
>T $D(LS)
Ø
>
```

*It's not there.*

Another point:

MUMPS, by convention, always interprets a nonzero number as a logically TRUE result and a zero number as a logically FALSE result.

Try using \$D on the data base. See what you can find out about it.

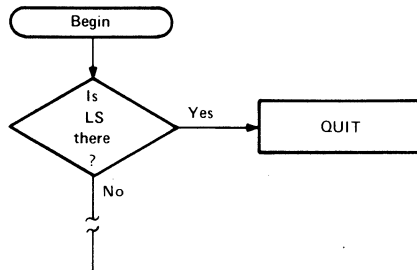
```
>T $D(AGE(.01))
_____
>T $D(SEX(.03))
_____
:
:
```

*zero or nonzero?*

Don't worry about the actual value returned by \$D when it's nonzero -- we'll tell you more about this later.



Back to the problem at hand.



11-1981

Now we can determine the existence of LS. Here's the first step of our data base initialization program.

```
I $D(LS) Q
```

*If LS is defined (i.e., TRUE), QUIT -- otherwise, continue.*

To say it another way:

- We know that when LS is defined, \$D(LS) returns a nonzero number.
- We also know that MUMPS interprets any nonzero number as being a TRUE value.

So, when \$D(LS) returns a 1 (for example), MUMPS reads the above command line as if it was

```
I 1 Q
```

*If 1, QUIT.*

and, of course, MUMPS QUITs because 1 is a TRUE value.

If LS is not defined, MUMPS reads the line as

```
I Ø Q
```

*IF Ø, QUIT.*

and the QUIT is not done since Ø is a FALSE value.

## HIGHER AND HIGHER

Now, before we return to the initialization program, there's one other function you should know about. The \$HIGH function. \$HIGH tells MUMPS to search an array and return the next higher subscript than the one specified. Try this:

*Let's create an array called XX for you to practice on.*

```
>S XX(0)=1,XX(2.9)=0.5,XX(131)=100,XX(132)=-5
```

```
>T $H(XX(0))
2.90
>T $H(XX(2.9))
131
>T $H(XX(131))
132
>T $H(XX(132))
-0.01
```

*Begin.*

*Here's the next higher subscript,*

*and the next,*

*and the next.*

*That's all - there aren't any higher subscripts.*

To find out if subscript 0 exists, type:

```
>T $H(XX(-.01))
0
>
```

*this can be any negative number*

The \$HIGH function is used to find the next numerically greater (higher) element in an array. \$H returns the actual value of the next higher subscript. A negative subscript value is used to determine the existence of subscript 0. If no higher subscript exists, \$H returns a -.01 .

Your turn. Try using \$H on our census data base. Try finding the lowest and highest subscripts in the NAM array. Then, see if you can write a program to find and store the highest subscript in the AGE array.

*HINT - you'll need to use two variables -- one to use with \$H and one to store the highest subscript when it is found.*

Here's the way we did it.

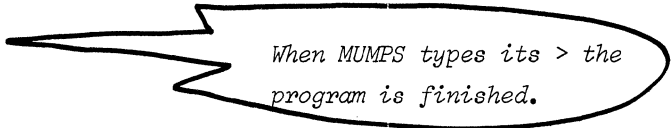
```
20.20 S X=-.01
20.30 S LS=X,X=$H(AGE(X)) I X<0 Q
20.40 G 20.3
```

Type it in! But first don't forget to FILE your version, if you write one. Then ERASE so that the steps of your program don't get mixed up with the steps in ours. Remember, even though you FILEd your program, a copy of it still remains in your program area.

Now DO our program. Type:

```
>D 20
```

```
>
```




*When MUMPS types its > the program is finished.*

Did it run ok? Any errors? Let's look at the variable LS to see if the program found the last subscript.

Type:

```
>I LS
```

```
>
```



*Is this the last subscript?*

Run your version now. But first, FILE ours, using the name INI (for initialize). Now ERASE your program area, LOAD your program, and DO it. When it's done, check the answer you got for the highest subscript. Does it match our answer? If it doesn't, check the programs for errors -- particularly typing errors. You shouldn't need any help from us.

Let's analyze how INI works.

Step 20.2 initializes temporary variable X to  $-.01$ . This value is used so that  $\$H(AGE())$  will start at subscript 0 (if it exists).

Step 20.3

- a. stores the contents of X in LS. This has to be done because X will eventually be set to -0.01 when \$H gets to the end of the array.
- b. stores the current subscript values produced by \$H in X.
- c. checks to see if X is a -0.01. When it is -0.01 we've reached the end of the array. Notice also that this could mean that there is no subscript greater than zero. In other words, there's no array.

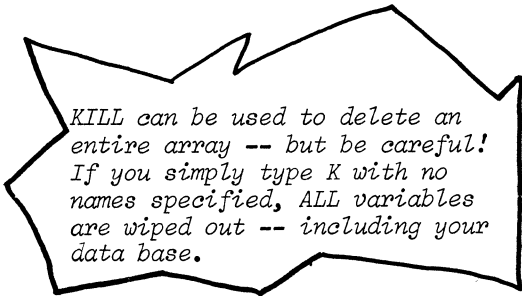
Now back to the original problem: how to write the data base initialization routine. Look back at our flowchart. We think you have enough information to do it now. So begin -- write the program yourself. For now, have it examine the test array XX so there'll be no risk of harming our census data. When you finish, FILE it, then DO it to see if it works OK. Try these tests.

1. First, KILL both XX and LS.

Type:

```
>K XX,LS  
>
```

Then run the program. Examine the contents of variable LS. It should contain a -.01.



*KILL can be used to delete an entire array -- but be careful! If you simply type K with no names specified, ALL variables are wiped out -- including your data base.*

2. Next, define a series of entries for array XX. For example:

```
>S XX(1)=1,XX(4.3)=4.3,XX(55)=55
```

Then run your program and examine LS again. It should contain the highest subscript of XX that you created.

Our turn.

Here's our version of the program.

```
20.10 I $D(LS) Q  
20.20 S X=-.01  
20.30 S LS=X,X=$H(XX(X)) I X<0 Q  
20.40 G 20.3
```

How does your program compare? Is it longer? -- shorter? Actually, all we did was add step 20.1 to the previous example that showed you about \$H. Step 20.1 is taken from the example demonstrating \$D.

Type the program in. DO it. Test it the same way you tested your program. Does it work? Good. Change XX to AGE and FILE it, using the name INI (for *INIT*ialize, of course). Now our census programs will always know where the end of the data base is without having to search for it each time.

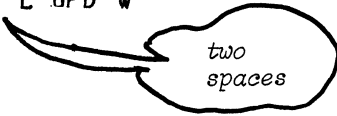
Here's how INI works:

- Step 20.1 checks to see if LS is defined. If it is defined, QUIT. We don't need to go further. If it is not defined, we'll go on.
- Step 20.2 initializes X. Since this step isn't executed unless LS is not defined, we'll define it by assigning X an initial value of -.01. As a convention, we'll use -.01 to tell our census programs that there is no data in the array. We'll assume further that since there is no data in the AGE array, none exists in the other arrays (SEX, NAM, OCC).
- Step 20.3 defines LS by SETting it to the value of X. Next, X is given the value of the next higher subscript. If X is less than 0 (\$H returned -.01 because there were no higher subscripts), we can QUIT -- LS now contains the value of the highest subscript. Otherwise, we'll go on counting.
- Step 20.4 completes the loop. Since X was not less than zero (X contains a subscript) GOTO 20.3 to assign the new value to LS and check for a higher subscript.

Now, our next task is to revise the existing census programs so that they use the data contained in variable LS.

Let's start with UPD. First FILE INI, then ERASE, LOAD UPD, and WRITE it out.

```
>E L UPD W  
>
```



*two spaces*

#### NOTICE

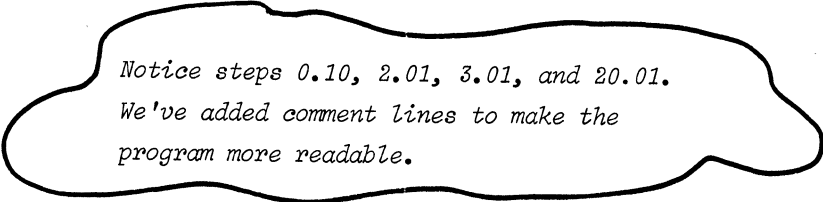
When a command has no arguments, two spaces must be used to separate it from the next command on the line. Remember, step and part numbers can be arguments to the ERASE command.

Examine it! Study it! Remember how UPD works? Here are the changes to be made:

- Eliminate commands which deal with the old \* dummy record.
- Use LS to keep track of the end of the data base.
- Update LS when additions are made.

You should be able to make all the changes yourself. Go ahead; flowchart it if you need to. When you've finished, give it a new name and FILE it.

Here's the way we did it. Type it in!



*Notice steps 0.10, 2.01, 3.01, and 20.01.  
We've added comment lines to make the  
program more readable.*

```

>E
>0.10 ;PART ONE DECIDES WHAT THE PROGRAM IS GOING TO DO

>1.10 T !,"CENSUS DATA UPDATE PROGRAM",!
>1.20 T !," OPTIONS: A=ADD DATA M=MODIFY DATA",!!
>1.21 D 20
>1.30 R " OPTION*",ANS I ANS="A" D 2
>1.40 IF ANS="M" D 3
>1.50 I ANS="" T !,"GOOD BYE",! Q
>1.60 G 1.2

>2.01 ;PART TWO IS FOR ADDING DATA
>2.10 R !," NAME:",NAM I NAM="" Q
>2.20 S LS=LS+.01,NAM(LS)=NAM
>2.30 R !,"AGE:",AGE(LS),!,"SEX:",SEX(LS),!," OCCUPATION:",OCC(LS),!
>2.40 G 2.1

>3.01 ;PART THREE IS FOR MODIFYING DATA
>3.05 I LS<0 T !," NO DATA BASE",! Q
>3.10 S N=LS R !!!," UPDATE DATA FOR:",!," NAME:",NAM
>3.20 I NAM="" Q
>3.22 I NAM(N)=NAM G 3.4
>3.30 S N=N-.01 I N<0 T !," NOT FOUND",! G 3.1
>3.35 G 3.22
>3.40 T !,"NAME(N)", " ",AGE(N)", " ",SEX(N)", " ",OCC(N),!
>3.50 R !,"CHANGES TO BE MADE",!," NAME:",NAM I NAM="" G 3.6
>3.52 S NAM(N)=NAM
>3.60 R !,"AGE:",AGE I AGE="" G 3.7
>3.62 S AGE(N)=AGE
>3.70 R !,"SEX:",SEX I SEX="" G 3.8
>3.72 S SEX(N)=SEX
>3.80 R !," OCCUPATION:",OCC I OCC="" G 3.1
>3.82 S OCC(N)=OCC G 3.1

>20.01 ;PART TWENTY INITIALIZES THE DATA BASE
>20.10 I $D(LS) Q
>20.20 S X=-.01
>20.30 S LS=X,X=$H(AGE(X)) I X<0 Q
>20.40 G 20.3
>

```

*This time, QUIT returns to step 1.3, not to the ">" that indicates Direct Mode.*

A semicolon (;) is used to begin a comment line. MUMPS ignores all characters that follow the semicolon. This lets you insert comments anywhere in a program to describe its various parts.

A QUIT in the range of steps specified by a DO terminates execution of any steps that remain. Control passes to the command immediately following the DO. QUIT outside the range of a DO stops the program. Control is returned to the terminal (MUMPS types a '>').

OK! Let's FILE the program. Use the name UD1. Then run it.

>F UD1 D 1

CENSUS DATA UPDATE PROGRAM

*Here we go.*

OPTIONS: A=ADD DATA M=MODIFY DATA

OPTION\*

*Select your option.*

*Add some data using the 'A' option, then examine it using the 'M' option. Change it. Remember what happens when you use a null string with the Modify option?*

When you're satisfied with the way the program works, stop it. Remember how? It's built right into the program. Remember what a null string is?

Now run your version of the program. Test it out. Does it do the job? If not, fix it; then FILE it. If it works OK, let's go on.

Our remaining task is to change the INP program. Remember what its function was? Right! It creates a new data base. But wait! We can do this job with the ADD option of our update program, UD1. Since we added the initialization routine that creates the LS variable, our program always knows where the end of the data base is -- even if the end and the beginning are the same place. Look back at step 2.2 of UD1. If the data base didn't exist, the initialization program would set LS to -.01. Our ADD program would simply add .01 to LS, which results in a zero value. This then becomes the value of the first subscript in the new data base.

So the INP program is obsolete! Delete it! It's no longer useful.

Type:

*Two spaces --  
remember?*

>E F INP

*Away it goes.*

>



## TIME FOR A CHANGE

Remember when you needed a permanent place to store your programs? We told you about the FILE and LOAD commands so you could store them in the disk memory. By now, we're sure you've discovered another need for permanent storage -- permanent data storage, of course. When you sign-off at your terminal by typing a HALT, the entire contents of your terminal's memory partition are wiped out. This includes not only any program that might have been there, but also all those local variables that contained our valuable census data. Local variables should be used only for temporary storage when our programs are running. Disk memory is the place to permanently store both programs and data.

The variables we've told you about up till now are classified as *local variables*. This means they are local to your partition -- they reside there. There's another kind of variable called a *global variable*, or simply *global*. These variables reside permanently on the disk memory. They can be used in MUMPS programs just like local variables. They also possess other qualities that we'll tell you about later on. The important thing for now is to know that globals can be used to permanently store our data base and any other data that we want to preserve.

Global variables are used just like any other variables. The rules are the same.

- Globals can be used in expressions.
- A global can contain up to 132 characters or a number as big as MUMPS allows.
- Globals can be TYPed, SET, and KILLed -- BUT THEY CANNOT BE USED WITH *READ*.
- Global variables can be subscripted.
- Global names are just like any other variable names -- they follow the same rules.

THE DIFFERENCE IS THAT GLOBAL VARIABLE NAMES ARE ALWAYS PRECEDED BY AN UP-ARROW (↑).

Here are some global names:

```
↑NAM
↑A
↑X32
```

Try this; type:

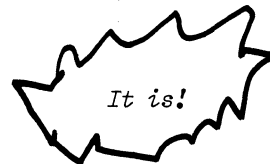
```
>S ↑A=" THE QUICK BROWN FOX JUMPED..."
>I ↑A
THE QUICK BROWN FOX JUMPED...
>
```

Log-out! Type:

```
>H
EXIT
```

Now log-in again. See if ↑A is still there.

```
MUMPS-11 V02 #6
UCI:
>I ↑A
THE QUICK BROWN FOX JUMPED...
>
```



Get rid of A. Type:

```
>K ↑A
>I ↑A
UNDEF>0 @
>
```



OK! Now it's time to get our census data storage on a better footing. We'll store our data base in *global variable arrays*. All you have to do is add an up-arrow (↑)

in front of every variable name that is to contain permanent data. In other words, change all

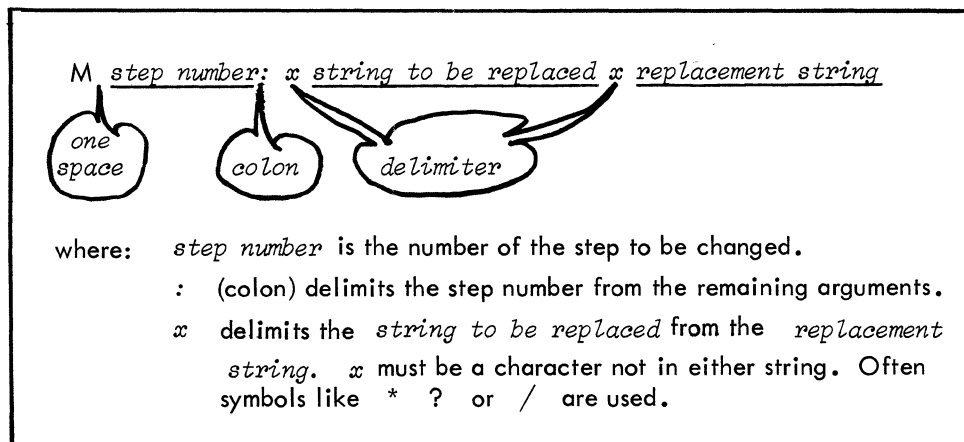
<u>occurrences of:</u>	<u>to:</u>
LS	↑LS
NAM(N)	↑NAM(N)
AGE(N)	↑AGE(N)
SEX(N)	↑SEX(N)
OCC(N)	↑OCC(N)

in the UDI program.

The other variables, like N, TMP, and X, should remain as local variables since they hold transient or intermediate information.

"Not again!", you say. No, we're not going to make you retype any more programs. You can use the MODIFY command to *edit* the steps of the MUMPS program that's in your partition. MODIFY lets you alter any number of characters in a program step.

Here's the general form of MODIFY:



MODIFY tells MUMPS to search the specified step for the first occurrence of the *string to be replaced* and substitute the *replacement string*. Once this is done, MUMPS types out the corrected step. If you don't specify a *string to be replaced*,

MUMPS puts the *replacement string* at the beginning of the step (i.e., right after the step number). If you don't specify a *replacement string*, the first occurrence of the *string to be replaced* will be deleted.



Here's UD1 again -- we've circled all the variables to be changed. Notice that we've also added a new step (2.35) to the program that uses local variables AGE, SEX, and OCC. This is necessary, since, if you'll remember, we can't READ directly into globals. So we've used some local variables instead, then put the data from there into the globals.

```

0.10 ;PART ONE DECIDES WHAT THE PROGRAM IS GOING TO DO

1.10 I !,"CENSUS DATA UPDATE PROGRAM",!
1.20 I !,"OPTIONS: A=ADD DATA M=MODIFY DATA",!!
1.21 D 20
1.30 R "OPTION*:",ANS I ANS="A" D 2
1.40 IF ANS="M" D 3
1.50 I ANS="" I !,"GOOD BYE",! Q
1.60 G 1.2

2.01 ;PART TWO IS FOR ADDING DATA
2.10 R !,"NAME:",NAM I NAM="" Q
2.20 S LS LS-.01,NAM LS =NAM
2.30 R !,"AGE:",AGE !,"SEX:",SEX !,"OCCUPATION:",OCC,!
2.35 S ↑AGE ↑LS =AGE, ↑SEX ↑LS =SEX, ↑OCC ↑LS =OCC
2.40 G 2.1

3.01 ;PART THREE IS FOR MODIFYING DATA
3.05 I LS <0 I !,"NO DATA BASE",! Q
3.10 S N LS R !!!,"UPDATE DATA FOR:",!, "NAME:",NAM
3.20 I NAM="" Q
3.22 I NAM N =NAM G 3.4
3.30 S N =N-.01 I N <0 I !,"NOT FOUND",! G 3.1
3.35 G 3.22
3.40 I !,NAM N," "AGE N," "SEX N," "OCC N,!
3.50 R !,"CHANGES TO BE MADE",!, "NAME:",NAM I NAM="" G 3.6
3.52 S NAM N =NAM
3.60 R !,"AGE:",AGE I AGE="" G 3.7
3.62 S AGE N =AGE
3.70 R !,"SEX:",SEX I SEX="" G 3.8
3.72 S SEX N =SEX
3.80 R !,"OCCUPATION:",OCC I OCC="" G 3.1
3.82 S OCC N =OCC G 3.1

20.1 I $D LS Q
20.20 S X=-.01
20.30 S LS X,X=$H AGE X) I X<0 Q
20.40 G 20.3

```



The first step to change is 2.2. LS appears three times. It has to be changed to ↑LS. Since slash (/) is not a character in the line, we'll use it as the delimiter.

Type:

```
>M 2.2: /LS/ ↑LS
```

```
>W 2.2  
2.20 S ↑LS=LS+.01, NAM(LS)=NAM
```

*WRITE out the step.*

```
>
```

*Fixed one LS*

This time we want to change the second occurrence of LS. If we use the same MODIFY command as before, MUMPS will simply add a second ↑ to the first LS -- like this:

```
2.2 S ↑↑LS...
```

That's not what we want. So think! What makes the second LS unique? The equal sign to its left, naturally. Now change it.

```
>M 2.2: /=LS/= ↑LS
```

```
>W 2.2  
2.20 S ↑LS= ↑LS+.01, NAM(LS)=NAM
```

```
>
```

*Fixed the second LS*

The last LS? Same story. What makes it unique? The left parenthesis, of course.

```
>M 2.2: /(LS/( ↑LS
```

```
>W 2.2  
2.20 S ↑LS= ↑LS+.01, NAM( ↑LS)=NAM
```

```
>
```

*That does it!*

We must also change NAM -- but just the first occurrence of it (i.e., NAM(LS)). We want the subscripted variable NAM(↑LS) to be a global variable. The simple variable NAM must remain in a local variable. This is also true of AGE, SEX, and OCC.

You're on your own. You change all the other variables we have circled. If you make a mistake, use MODIFY to correct it. When you're finished, FILE the program as UD2. That way you'll have UD1 as a backup copy of the program.

The only task that remains is to transfer the census data now in the local variable arrays to global arrays. You could write a very short program to do that, particularly if our LS variable (Last Subscript, remember?) is defined. See if it's there.

Type:

```
>T LS
```

```
>
```

*Is it there? Did you get an error, or a number?*

If you get an error message, LS isn't there. LOAD UD1 and DO it so the initialization routine can create LS and calculate the correct value for it.

OK. Now you can write the data transfer program, but don't forget to ERASE first. See how concisely you can write the program.

Did you write it like this?

```
>1 .1 S I=0
>1 .2 S ↑NAM(I)=NAM(I), ↑AGE(I)=AGE(I), ↑SEX(I)=SEX(I), ↑OCC(I)=OCC(I)
>1 .3 I I=LS S ↑LS=LS Q
>1 .4 S I=I+.01 G 1.2
>
```

Good. This will work OK, but we found a shorter way. Here it is.

*This is the FOR command.*

```
1.10 F I=0: .01: LS D 2
2.10 S ↑NAM(I)=NAM(I), ↑AGE(I)=AGE(I), ↑SEX(I)=SEX(I), ↑OCC(I)=OCC(I)
2.20 S ↑LS=LS
```

See how many lines we eliminated. Of course, we had to cheat a little -- we used a new command back at Step 1.1.

```
1.10 F I=0:.01:LS D 2
```

It says:

"FOR I equal to an *initial value* of zero and subsequently *incremented* by .01 until a *limit value*, contained in LS, is reached, DO all the steps in part 2."

In other words, if LS contained the value .10, then FOR would cause I to have eleven different values -- from 0 through .10. FOR each value of I, part 2 would be DOne.

The FOR command makes it easy to create program loops. Up till now, you've been using SET, IF, DO and GOTO to *iterate* (i.e., loop). Look back at the first form of the transfer program.

- Step 1.1 SETS I to an *initial value*.
- Step 1.4 checks to see IF I has reached the *limit value* (LS).
- Step 1.5 increases I by an amount equal to the *increment value*.

FOR does all this in one command. This means your program:

- contains fewer steps and commands
- takes less space in memory and therefore runs faster

To be more precise, the general form of FOR is:

FOR *local variable* *one space* *equal sign* *initial value* *colon* *increment value* *limit value* *one space* *next command(s)*

The diagram shows the general form of the FOR command: FOR local variable initial value: increment value limit value next command(s). Callouts in speech bubbles point to the punctuation: 'one space' for the space after 'FOR', 'equal sign' for '=', 'colon' for ':', and 'one space' for the space after 'limit value'. Brackets below the text group 'initial value', 'increment value', and 'limit value' together, and 'next command(s)' together.

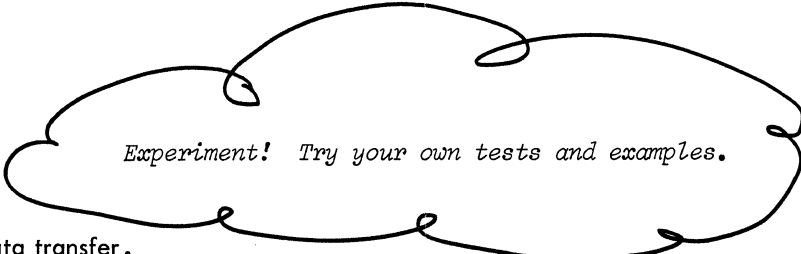
*Initial value, increment value, and limit value can be any number, variable or expression that yields a legal MUMPS number (i.e., ±21474836.47).*

This is the command or commands that are to be done as part of the loop.

Try this:

Type:

```
>F I=1:1:100 I " TEST NUMBER ",I,!
TEST NUMBER 1
TEST NUMBER 2
TEST NUMBER 3
TEST NUMBER 4
TEST NUM ...
.
.
.
TEST NUMBER 100
>
```



*Experiment! Try your own tests and examples.*

Back to data transfer.

Let's transfer the census data to disk memory. Enter and FILE the program you want to use. Ours or yours -- or both if you wish. Then run it.

Still your turn.

You've got census data stored on the disk now, and our update program, UD2, can maintain the data base. Run UD2 for a while. Add some new data of your own -- change some existing data. Exercise the program -- test it out. Log-out!

- Log-in again.
- Start up UD2 again.
- Modify some data. It's all there. Right?

#### A PARTING THOUGHT

Here's something for you to think about while you catch your breath before starting the next chapter.



Why are we gathering all this data? Of what use is it? To get census *information*.

Information that supplies answers to questions like:

- What's the average age?
- What's the female to male ratio?
- How many people between ages 30 and 32 are women?

Why don't you try to write a program that will tell us how many people in our data base are 30 years old?

## CHAPTER 6

# FORM FOLLOWS FUNCTION

Did you get the information you wanted? How many 30-year-old people were there in the data base?

Did your *retrieval program* work? The program to compute the number of 30-year-olds, that is.

Here's the way we wrote it:

```
1.10 R !,"AGE:",X,! I X="" T !,"GOOD BYE",! Q
1.20 I X?D,X>0,X<100 G 2
1.30 T !,"NOT VALID NUMBER",! G 1.1

2.10 D 3 T !,"THERE ARE ",TOT," PEOPLE AGE ",X,! G 1.1

3.10 S TOT=0,I=-.01
3.20 S I=$H(↑AGE(I)) I I=>0,↑AGE(I)=X S TOT=TOT+I
3.30 I I=>0 G 3.2
```

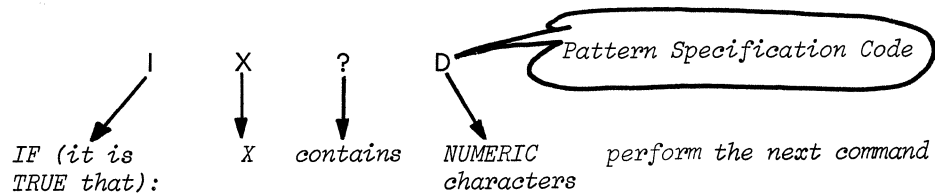
Though your version of this program may not look like ours, it should do essentially the same thing. We wrote our version so that it would work with any age between 1 and 99.

Here's a brief analysis. The program is divided into three functional parts.

- Part 1 inputs and verifies the age for which the search is to be done. Any age other than a number between 1 and 99 or a null string is ignored and the message NOT VALID is typed.
- Part 2 obtains and outputs the results.
- Part 3 searches and counts each entry of the specified age in the ↑AGE array. When  $I \leq 0$ , control returns to 2.1.

Look at step 1.2 closely. Do you see something you don't understand?

X?D illustrates the use of a special feature of MUMPS called pattern verification. This lets your program examine strings for specific character patterns.



Pattern verification is used in logical expressions, that is, expressions that produce a TRUE or FALSE result.

- Pattern Verification -

The string variable preceding the ? is examined for the occurrence of the character patterns specified by Pattern Specification Code (psc) to the right of the ?. If a matching condition exists, the result is TRUE; otherwise the result is FALSE. Codes can be grouped in any combination. Each code can be preceded by a number (0 - 9) to specify the number of occurrences of a particular character type. If zero is specified, the associated character type is ignored. If no number is specified, an indefinite number of characters of the specified type is accepted.

Here is a list of the basic Pattern Specification Codes:

- A    Verify capital letters.
- D    Verify digits.
- P    Verify punctuation (i.e., anything other than letters or numbers).
- W    Verify any character.

Back to our program.

You should be able to understand how the program works now, so type it in and DO it. REMEMBER TO ERASE FIRST.

```

>E
>I .10 R !,"AGE:",X,! I X=....
>I .20 I X?D,X....
.
.
.
.
>D I
AGE:300
NOT VALID NUMBER
AGE:AB
NOT VALID NUMBER
AGE:30
THERE ARE 8 PEOPLE AGE 30
AGE:

```

*Try age 30.*

*Your turn.*

Had enough? OK, stop the program! Type a null string when it asks for a new age.

```

AGE:
GOOD BYE
>

```



Now FILE the program -- we'll be using it later. Call it A1.

```

>F A1
>

```

WHAT IF . . .

How well did our program work? Did you have to wait long for the results? Chances are that you didn't wait too long for the answer. Your data base is probably small. Most likely it contains less than 100 entries. Not very many. How big will the data base ultimately be? How many people will we have census information on?

- One thousand?
- Ten thousand?
- One hundred thousand?
- Millions?

Pick a small town, say a town between one thousand and ten thousand people. Our data base could be a lot bigger than it is now. What if our data base had 100 times more entries than it currently has? How long would it take for our A1 program to do its work?

Let's run some tests to find out. First, we'll measure the time it takes to run A1. To time the event, we'll need a timer. MUMPS has a timer called the \$T System Variable. This is a special variable that's built into MUMPS itself. It contains a number between 0 and 86,399. This number is the total number of seconds elapsed since midnight.

\$T is incremented automatically by MUMPS each second.

\$T can be examined by using TYPE or by SETting another variable equal to \$T. Since \$T is a System Variable, MUMPS can't let you or your program change (SET) it.

Try this:

```
>S $T=200
```

```
SYNTAX>0 @
```

```
>I $T
```

```
47552
```

```
>
```

*Syntax error. You can't set \$T.*

*Here's the current value.*

Do it again.

```
>I $T
```

```
47574
```

```
>
```

*Here's the current value.*

*The difference between the first and second values is the number of seconds elapsed between the first interrogation and the second interrogation of \$T.*

Now to our test. Since part 3 of the A1 program is the part that's retrieving the global data, we need only measure the time it takes to run. What we must do is write a program that:

- Remembers the value of \$T when part 3 begins running .
- Remembers the value of \$T when part 3 is through .
- Calculates the time required to do part 3 .

Let's begin .

Make sure A1 is there, in your partition. Try WRITing it out! If it's not there, LOAD it.

```
>L A1
>
```

Here's the new part that does the timing. Type it right in!

```
>99.05 K TOT,B,T,LPT,N
>99.10 R !," NUMBER OF ITERATIONS=",N,!
>99.20 S T=$T F B=1:1:N D 99.6 Save start time.
>99.30 S LPT=$T-T
>99.40 S T=$T F B=1:1:N D 3 DO part 3.
>99.50 T $T-T-LPT/N," SECONDS",! Save finish time and type
>99.60 Q out elapsed time.
>
```

This program is somewhat more complex than that we outlined earlier. Since \$T is only accurate to .5 seconds, we must time part 3 a number of times to get a reasonably accurate answer. This answer represents the average time required to do part 3. We must also eliminate time consumed in activities not related to part 3. Here's how it works:

Step 99.05 KILLS all local variables that are created (SET) during the operation of part 3 or part 99.

Step 99.1 Requests the number of iterations of part 3 (variable N)

Steps 99.2 and 99.3 Measure the time required to do the FOR and DO commands for the specified number of iterations (variable LPT).

Step 99.4 Does part 3 for the specified number of iterations.

Step 99.5 Calculates the time required to do part 3 and types out the value.

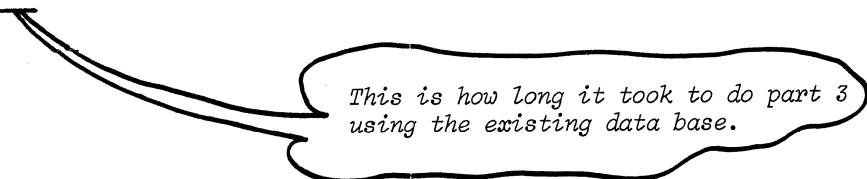
Now, since we want to time only part 3, we won't need to run the entire program -- just part 3 and the new part 99.

We're almost ready to start. Just one more thing. Since we're not running part 1, the variable X won't get defined and we'll get an UNDEF error when part 3 runs. So SET X to some age, say 30.

```
>S X=30  
>
```

Ready to begin. Let's start timing. Type:

```
>D 99  
NUMBER OF ITERATIONS=10  
SECONDS  
_____
```



*This is how long it took to do part 3 using the existing data base.*

Let's FILE this version of A1 with the new part 99 built in. We'll need it later.

```
>F A1  
>
```

It's reasonable to expect that if our data base increases in size, the time required to search it will also increase. As an exercise, you may wish to use a larger data base to see how much longer our access time will be. Rather than typing in new data, simply write a program to replicate the existing age array (since that's the only global part 3 is concerned with). All it takes is a FOR loop. Also, make sure ↑LS is updated by your program when it's done. If you try to replicate our current data base (30 entries) more than three or four times, you'll have a long wait during both the replication and the timing.

Observe how part 3 of our program works. It must examine each entry in the ↑AGE array. When there are more entries it takes longer to search. We want our census data to be *retrieved* as quickly as possible -- that's one of the reasons we're using a computer.

Notice how the four arrays in the data base are arranged.

SUBSCRIPT	↑AGE	↑SEX	↑NAM	↑OCC
.00	46	M	HENRY ADAMS	CARPENTER
.01	15	M	BILL SMITH	STUDENT
.02	30	F	ALTHEA BROWN	CHEMIST
.03	22	M	PAUL JOHNSON	PROGRAMMER
.04	86	F	JUDY ZWINK	GRANDMOTHER
.nn				

11-2003

Is there any particular order to it? Yes! The data exists in four parallel arrays -- in the order in which it was first input. It was not arranged according to any particular characteristics of the data itself. This organization doesn't seem to be particularly useful in obtaining census data based on age. Our program must examine each ↑AGE entry to find the right ones. If the age data were ordered in some more significant way -- say sequentially by age -- our program could work faster. It could simply go to the place where, for example, the age 30 entries were stored, and count them. The result could be obtained in much less time.

*The point is that the organization or arrangement of a data base can significantly affect its usefulness for retrieval. A data base should be designed to suit the way it's to be used.*

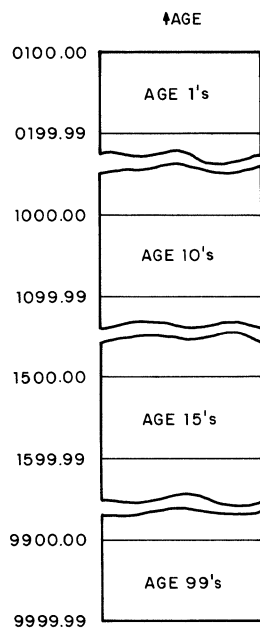


Effectively, our program must select all the entries of the same age in order to count them up. Each time we run the program for the same age group, more time is wasted because the group is sorted over again. How can we avoid this time-wasting duplication of effort? One way would be to simply sort the census data into age groups when it is first input to the data base. We would put all age 1's together, age 2's together, ..., age 99's together. How can the data base be arranged to accomplish this? How many ways can you think of?

We can think of at least three ways.

### ONE WAY

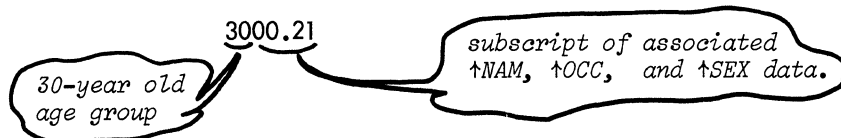
Rearrange  $\uparrow$ AGE so that all ages are grouped in a predetermined range of subscripts. Like this:



11-2004

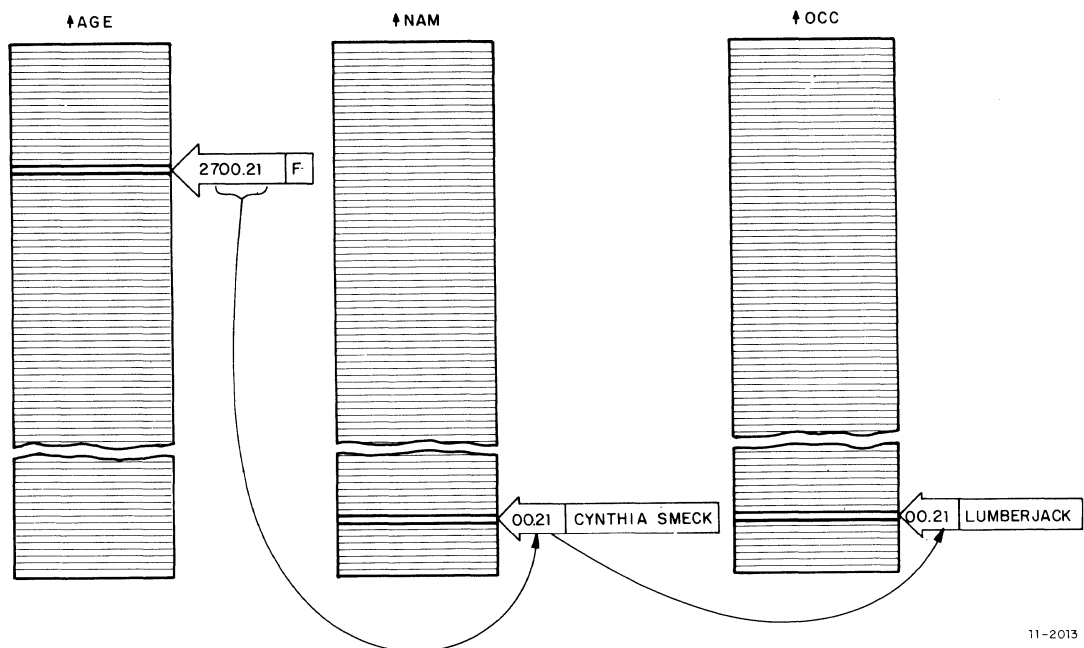
Each age group would have a predefined area determined by its subscripts. All age 1's would be assigned subscripts in the range 100.00 to 199.99; age 15's: 1500.00 through 1599.99; age 99's: 9900.00 through 9999.99. Since we can tell age from the two most significant (leftmost) digits of any subscript in  $\uparrow$ AGE, we can easily count up all entries in a particular group. The remaining digits could be the subscript values used in the associated  $\uparrow$ OCC,  $\uparrow$ NAM, and  $\uparrow$ SEX arrays.

Thus, an ↑AGE subscript would be interpreted as:



Further, age no longer needs to be stored in the variable itself, since the subscript itself is the age value. Now, other data could be stored there -- like SEX, for example. We could then eliminate the ↑SEX array and save disk memory space.

A subscript such as 2701.20 would locate data in this kind of data base as follows:



11-2013

A program to search an array like this might be:

```
15.10 S P=Z*100, TOT=0 Z contains the first subscript in the age group to
      be searched. To search for age 10's, P = 1000.00;
      age 35's, P=3500.00, etc.
```

```
15.20 S P=$H(↑AGE(P-.01)) I P<0 Q
15.25 I P>(Z*100+99.99) Q
15.30 S TOT=TOT+1 G 15.2
```

- Remember -

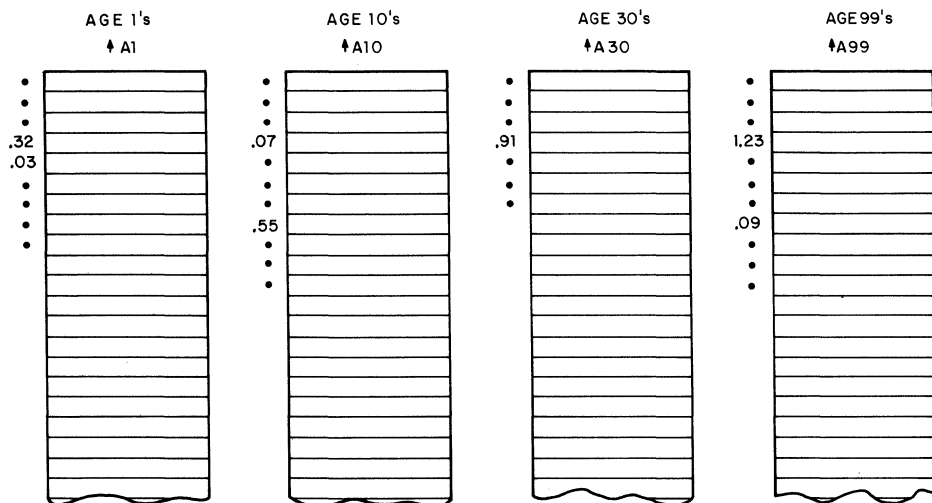
*\$HIGH returns a -.01 when there are no higher subscripts. If P were ever less than zero, that would mean that there were no entries with age over Z, and maybe none with age Z.*

There are, however, some disadvantages to this particular data arrangement.

- o If the census data is to increase with population growth, all available subscripts will eventually be used up. When that happens, the data base will have to be reorganized to incorporate longer subscripts.
- o Eventually the legal subscript limits could be exceeded.
- o It's possible to have subscripts in ↑AGE like: 2700:20, 1300.20, 4500.20, etc. This replication of the lower four digits would doubtless be disastrous for you. However, it's unlikely that this would happen in a normal situation, since these lower four digits are derived directly from the subscripts in ↑NAM and ↑OCC.

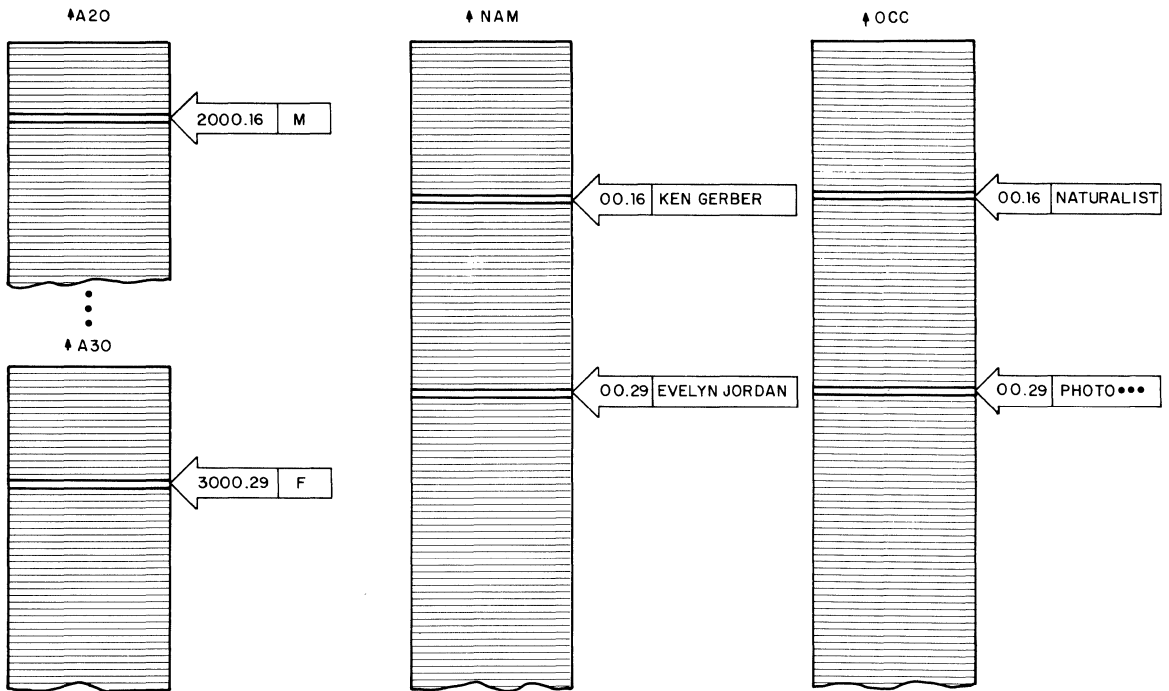
### ANOTHER WAY

Create a separate global array for each age group.



11-2005

Entries in each array could contain the SEX data as before, and the subscripts would still key into ↑NAM and ↑OCC arrays.



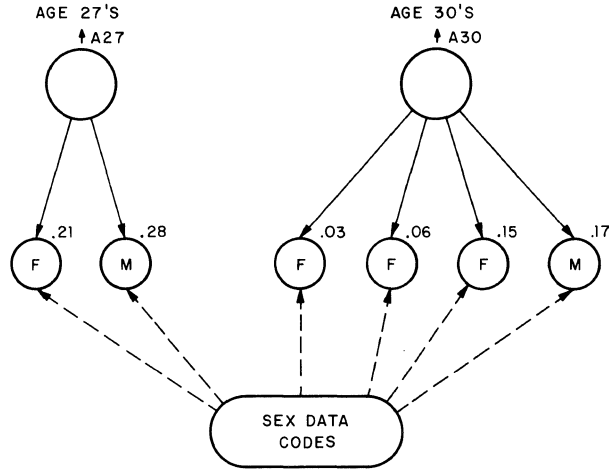
11-2012

This approach can be a very useful way of storing census data to facilitate data searches based on age. To do this, however, requires some relatively sophisticated programming techniques to keep track of all the names of the age globals. We're not ready to tell you about them yet!

But there is a way that we are ready to tell you about.

THE WAY

We'll use some of the ideas shown in the last example, but rather than having separate globals for each age group like ↑A27 and ↑A30, (below),

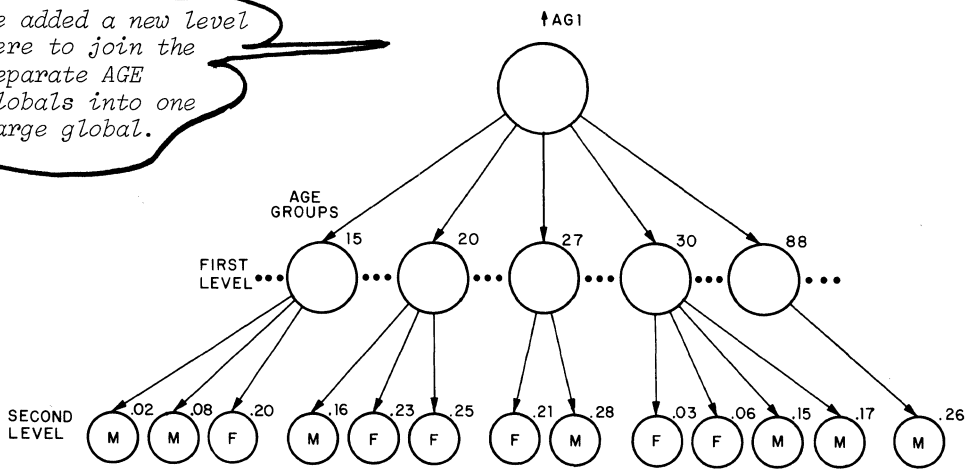


11-1982

*Here's a new way to diagram global structure. MUMPS programmers normally use this method.*

we'll combine them into one big global, like this:

*We added a new level here to join the separate AGE globals into one large global.*



11-1988



Similarly

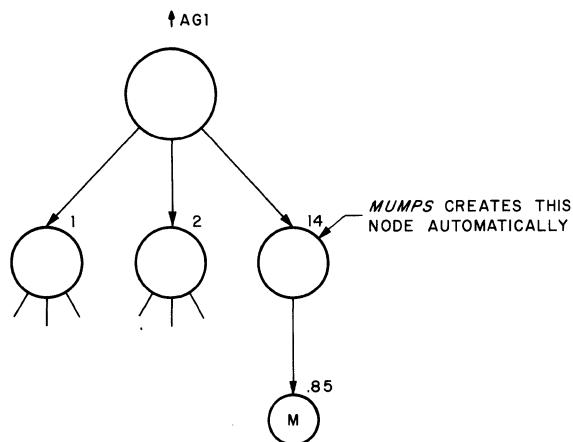
`↑AG1(15,.08)` refers to a node in the age 15 group that contains the sex code "M".

Global array nodes are created simply by defining them with the SET command. When more than one level of subscripting is used, MUMPS automatically defines all intermediate levels necessary to reach the desired level. Therefore, if we want to create a new entry in our global for a new age group, say age 14, with .85 as the second level subscript and SEX code M

`S ↑AG1(14,.85)="M"`

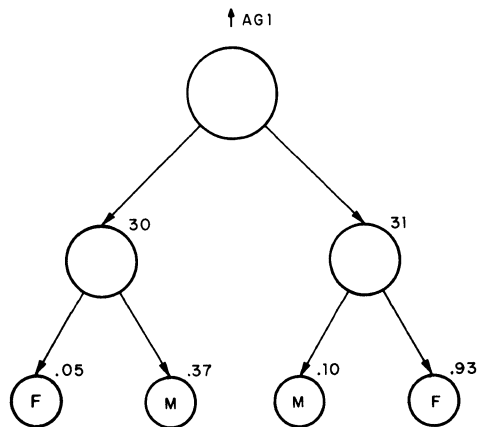
would do it.

MUMPS adds a new "branch" to our global "tree" like this:



11-1989

Like other variables, global nodes can be deleted using the KILL command. For example, if part of our age global looked like this:

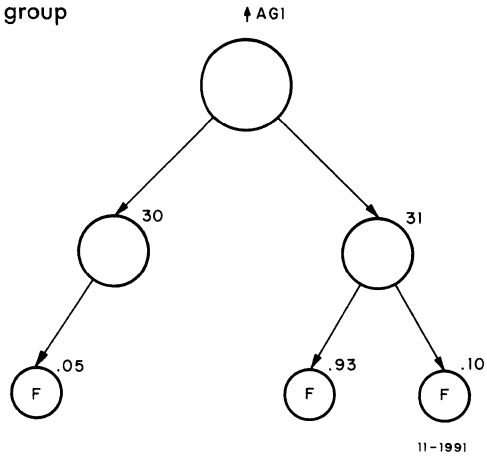


11-1990

we could:

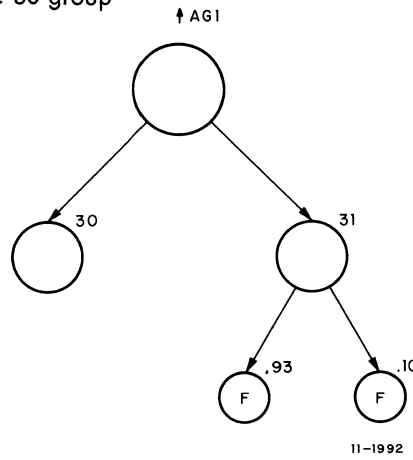
- Delete the .37 entry in the age 30 group

**K ↑AG1(30, .37)**



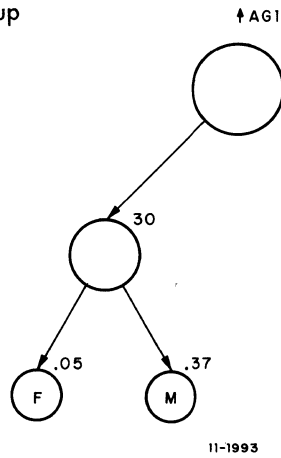
- Delete the .05 and .37 entries in the age 30 group

**K ↑AG1(30, .05), ↑AG1(30, .37)**



- Delete all entries in the age 31 group

**K ↑AG1(31)**



- Delete the entire global array

**K ↑AG1**



So! Now that you know something about the nature of global arrays, let's convert the old  $\uparrow$ AGE and  $\uparrow$ SEX arrays into one 2-level global called  $\uparrow$ AG1. Use age data for the first subscript and the old array subscript for the second, as we described earlier. If you think you can write the program yourself, go ahead.

Here's our way. Just two commands on one line!

Try it:

```
>F I=0:.01:↑LS S ↑AG1(↑AGE(I),I)=↑SEX(I)
```

```
>  all done
```

Here's how it works. First we create a FOR loop that generates all subscripts that exist in our data base (  $\uparrow$ LS contains the upper limit). Then, FOR each subscript value, we create a new array variable. The first subscript of the variable is the age value contained in  $\uparrow$ AGE(I); the second subscript is the value of I itself. This new variable contains the sex code (M or F) of the associated  $\uparrow$ SEX(I) array entry.

Now the data base is arranged in a way that should significantly reduce the time required to retrieve data by age group.

How much faster?

Let's see how much less time is required to find out how many age 30 are in the census data now. Load A1, the retrieval program that we showed you in the beginning of this chapter. Remember?

```
>L A1
```

```
>
```

Now, since A1 was written to access the old  $\uparrow$ AGE array, we'll have to change it so that it will work with the new  $\uparrow$ AG1 array. Simply replace part 3 with this new part 3.

```
3.10 S TOT=0,I=-.01  
3.20 S I=$H(↑AG1(X,I)) I I=>0 S TOT=TOT+1 G 3.2
```

```
>
```

Write out the whole program.

>W

Examine it. Make sure there aren't any errors. Do you understand how this new part 3 works?

Give up?

To begin with, this part assumes that X has already been set (in part 1) to the age value for which the search is to be performed.

Step 3.1 SETS the counter TOT to 0 and I to  $-.01$  so that the \$H in the next step will begin at subscript zero.

Step 3.2 SETS I to the value of the next higher subscript. As long as I is greater than or equal to zero, then \$H has told us that there is a higher subscript. Therefore, count it and loop back to the beginning of the step. When \$H returns a  $-.01$ , there are no higher subscripts, therefore no more entries in this age group. The loop is broken and, since there are no higher steps in this part, the operation is complete.

Part 3 stops only when \$H causes I to become negative. Occasions when this happens are:

- 1) when there is no node for any age group;
- 2) when there is no node for a given age group;
- 3) when all nodes have been counted.

OK! Run the program for a while. See if it works. Test it out. When you're satisfied that it's working properly, FILE it as A2.

>D 1

AGE:30

THERE ARE PEOPLE AGE 30

AGE:

RE-TURN

Stop A2.

GOOD BYE

>F A2

File it.

Is this the same number you got when you ran AG1 for this age? If it isn't, there's something wrong with either the program or the data.

HOW LONG?

Now that the scene is set -- back to our timing test. Do you still have the results you got from the last time you ran A1? If not, run A1 again.

S X=30 L A1 D 99

our standard test age

NUMBER OF ITERATIONS=10  
SECONDS

>

There's the old time.

Now for the new data base.

>L A2 D 99

NUMBER OF ITERATIONS=10  
SECONDS

>

There's the new time. Quite a noticeable difference.

See how much faster data retrieval can be when the data is more suitably organized?

The point of this whole chapter is, in large part, to start you thinking about how the size and shape of a data base can affect its usefulness. You'll be learning more about this in the next chapter.

Can you think of other census data retrieval applications that would work well with the current data base organization?

- How about changing A2 so it will tell us how many men or women are in a particular age group.
- Or you could try to find an even faster way for A2 to obtain the age totals. (Hint - eliminate calculation from A2 and add a new piece of data to our present structure.)

But remember, when you're planning a program, keep in mind the data base organization you're going to use and how your program can best work with it.



## CHAPTER 7

# INSIDE GLOBALS

In case we didn't tell you before, MUMPS is a timesharing system. MUMPS shares its time among a number of users and their programs. Although it appears that MUMPS is giving all its attention to you and your programs, it's really doing the same for others like yourself.

Because MUMPS is a timesharing system, the disk memory that stores programs and global data is also being shared. Everyone's programs and globals reside there.

Each time a program makes reference to a global variable,

```
S ↑AGE=X
```

```
T $H(↑SEX(I))
```

```
I $D(↑NAM(I,Y))
```

MUMPS must interrogate the disk memory. When it's doing this for your program, it can't be doing it for other programs. Therefore, other users' programs must wait. When another user's program is using the disk, your program may have to wait. How long? It depends ... on how many programs are queued-up waiting for the disk, how many interrogations (accesses) each program makes, and the *way* each program accesses the disk. Which brings us to the point of our discussion. The way a program uses the disk (i.e., the way it makes global references) can affect not only the amount of time required to access a global, but also the time spent by other programs waiting to use the disk.

Up till now, you've been making global references in your programs using what is called a *full global reference*. Full global simply means that a global reference contains all the information necessary to completely describe the global node. Like this:



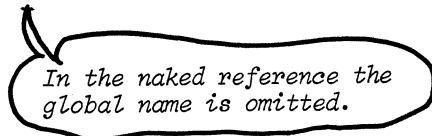
Whenever you use a full global reference, MUMPS looks up *all* the data required: to find the node you've specified, *even though* your program may have just accessed this global or even the same node many times before. Believe it or not, a lot of time can be consumed in this non-productive activity.

"Can we avoid this waste?" you ask. Yes, indeed! We'll give MUMPS only the information it really needs to do the job.

Once we've told MUMPS the name of the global that we're going to use, we don't have to tell it again as long as we continue to access nodes at the same or a lower subscript level (i.e., nodes with the same or greater number of subscripts). MUMPS "remembers" the last global access that you or your program made... all you need do is tell it the next subscript to be accessed, either at the same or a lower level.

The type of global access we're talking about is called the *naked global reference* or simply *naked reference*. This is done by using an abbreviated form of the full global reference. Here's what it looks like:

Naked Reference:

↑(X)  
  
In the naked reference the global name is omitted.

Here's the full reference again:

↑AGE(X)

Here's the way it works:

When you use naked reference, MUMPS assumes the name of the global you wish to access is the one given in the last full reference.

It further assumes that you wish to access nodes that are either at the same subscripting level reached by the last global reference or at a lower level.

To gain access to a global via naked reference, at least one full global reference must be made first.

MUMPS replaces the last subscript of the last reference (naked or full) with the first subscript of the naked reference. For example, if our last global reference was:

```
>S X=↑AGE(.29)
>
```

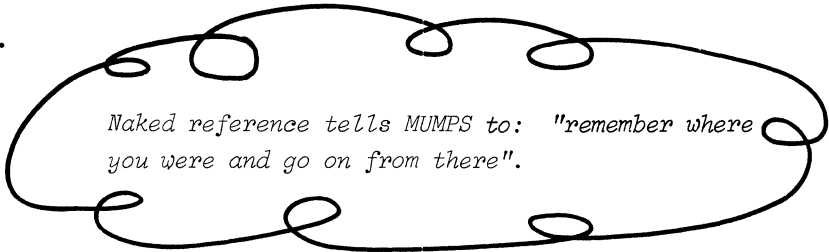
we could access ↑AGE(.20) like this:

```
>S X=↑(.20)
>
```

If ↑AGE(.02) is the next node to be accessed, we could write:

```
>S X=↑(.02)
>
```

and so on.



On the other hand, if we tried to do a naked reference without a prior full global reference:

```
>↑(203)
UNDEF>0 @
>
```

an UNDEF error message results. Try for yourself - let's use the naked reference to TYPE out some names from the ↑NAM global.



```
>T ↑(.04)
22
>
```

*Oops! We forgot to tell MUMPS the global's name. Since we did make a full global reference to ↑AGE (on the previous page), MUMPS assumed that we wanted to access that array.*

```
>T ↑NAM(.04)
PAUL JOHNSON
>
```

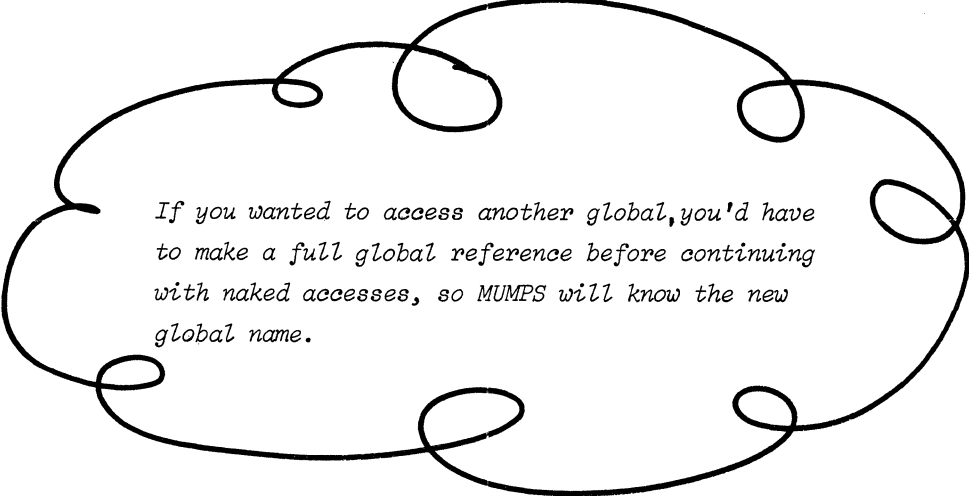
```
>T ↑(.10)
IAN MCKENZIE
>
```

```
>T ↑(.02)
BILL SMITH
>
```

```
>T ↑(.28)
KEN MASER
>
```

#### THE GOLDEN RULE OF NAKED REFERENCE

Naked reference is logical only when you know where you are in the global data base.



*If you wanted to access another global, you'd have to make a full global reference before continuing with naked accesses, so MUMPS will know the new global name.*

Now let's see what the naked reference can do for us. Let's find out how much faster our programs can run. First, we'll try it with the A1 program that searched the linear (i.e., single subscripted) global ↑AGE. So LOAD and WRITE out part 3 of A1 (this is the only part that accesses global data).

```
>L A1 W 3
3.10 S TOT=0,I=-.01
3.20 S I=$H(↑AGE(I)) I I=>0,↑AGE(I)=X S TOT=TOT+1
3.30 I I=>0 G 3.2
```

How can A1 be changed to do naked accesses? The most obvious change is to step 3.2. MODIFY 3.2 so that the two global references are naked references rather than full global references. It should look like this:

```
3.20 S I=$H(↑(I)) I I=>0,↑(I)=X S TOT=TOT+1 G 3.2
```

Now we have to add a full global reference to allow the naked references to work. The best way to do this is to change 3.1 so that it does one complete data access before entering the loop formed by steps 3.2 and 3.3. Here's the new step 3.1:

```
3.10 S TOT=0,I=$H(↑AGE(-.01)) I I<0 Q
```

Then add this new step:

```
3.15 I ↑(I)=X S TOT=1
```

Here's how this new part 3 works:

- Step 3.1 first initializes the counter TOT; then the subscript of the first node in the array is obtained using \$H with a full global reference; last the value returned by \$H is examined to see whether there was a HIGHer subscript.
- Step 3.15 uses a naked reference to verify that the contents of the first node accessed matches the requested age (variable X). If there is a match, the counter TOT is set to one.
- Step 3.2 and step 3.3 are essentially identical to our original version of the program (Chapter 6) except that all global references are naked. After the first access and verification sequence (steps 3.1 and 3.15), all subsequent global accesses are performed in the loop created by these steps. When all nodes in the array have been accessed, \$H sets I to a negative value and the IF test in step 3.3 fails, thus terminating program execution of this part.

Now, let's run the program. Does part 3 take less time to execute than before? Once again, we'll use the time routine we put in part 99, so manually set X to the 30-year age value, and start up the program.

```
>S X=30 D 99
NUMBER OF ITERATIONS=10
SECONDS
>
```

Next, change the A2 program, the one that uses the 2-subscript-level global AG1, so that it uses naked global references.

*Don't forget to FILE our modified version of A1 first.  
Use the name A3.*

Load A2 and WRITE out its part 3 now.

```
>L A2 W 3
3.10 S TOT=0,I=-.01
3.20 S I=$H(↑AG1(X,I)) I I=>0 S TOT=TOT+1 G 3.2
>
```

Now let's modify it to use naked references. Like this:

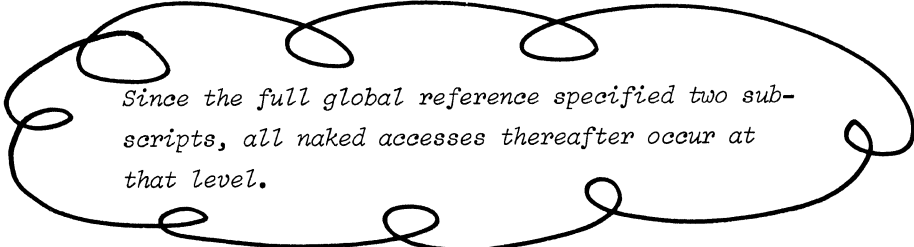
```
3.10 S TOT=0,I=$H(↑AG1(X,-.01)) I I<0 Q
3.15 S TOT=1
3.20 S I=$H(↑(I)) I I=>0 S TOT=TOT+1 G 3.2
```

This part 3 works in a manner very similar to that of the previous one.

- Step 3.1 initializes the counter TOT and gets the subscript of the first node. If \$H returns a  $-.01$ , the node is not there and we QUIT.
- Step 3.15 Since we didn't QUIT, we'll count this first node.
- Step 3.2 searches for and counts all succeeding nodes looping on itself until \$H returns the  $-.01$ .

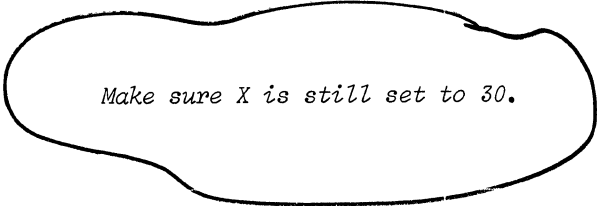
The interesting difference between this part 3 and the part 3 in A3 is that we're now dealing with a 2-level global. Notice how naked global reference works here.

In step 3.1, using a full global reference, we obtain the subscript of the first node at the *second* level. Now, the naked reference in step 3.2 will obtain the next higher node (\$H) at the second level and will continue to do so for each iteration of the loop.



*Since the full global reference specified two subscripts, all naked accesses thereafter occur at that level.*

Now run this program. See if naked accesses make retrieval faster when there's a 2-level global.

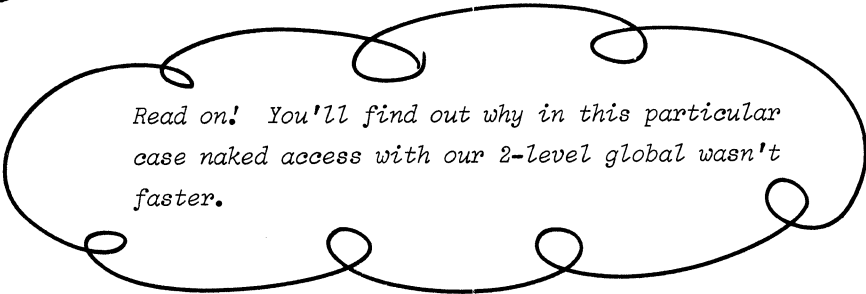


*Make sure X is still set to 30.*

```
>T X  
30  
>
```

If it's not, set it. Also file this version of our program as A4 . Then it's ready to run.

```
>S X=30 F A 4 D 99  
NUMBER OF ITERATIONS=10  
SECONDS
```



*Read on! You'll find out why in this particular case naked access with our 2-level global wasn't faster.*

## LOGICAL-TO-PHYSICAL

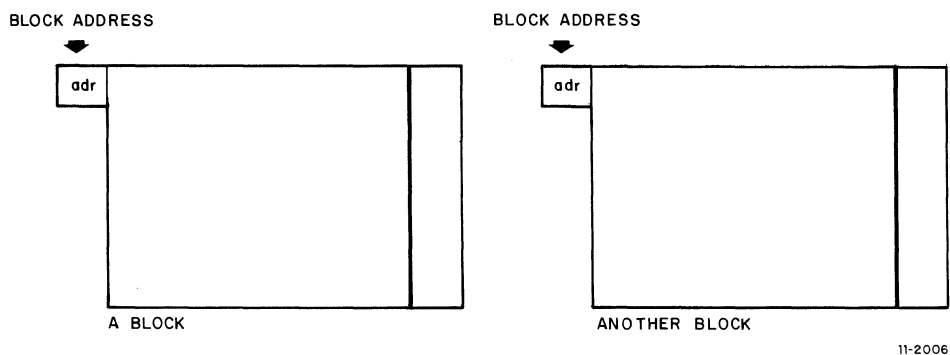
Up till now, we've told you how to incorporate naked references in your programs and how naked references *appear* to work.

Naked references allow MUMPS to obtain global data more rapidly by eliminating unnecessary disk memory accesses.

We say "appear" because the way naked accesses physically operate is different from the way they logically appear to operate. In order to use the naked reference correctly and effectively, it is important for you to understand:

- o the physical organization of global data
- o the physical operation of both full and naked references

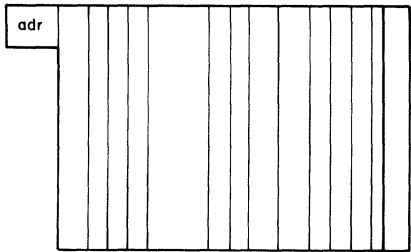
The disk memory that MUMPS uses is divided into physical storage areas called blocks. Data stored in the disk memory resides in these blocks. Each block has a unique identification number called the block address. When MUMPS stores or retrieves data on the disk, it is writing or reading specifically addressed blocks. Communication between MUMPS and the disk is performed one block at a time.



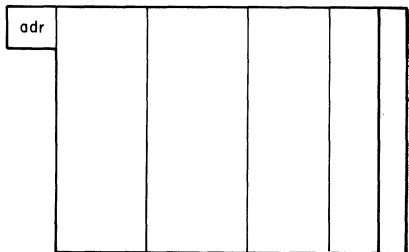
Since a block is a physical unit, the amount of data it can hold is limited. A block can hold a large number of short data items or a small number of long data items. Global nodes (or variables) are the data items stored in blocks. A node residing in a block is usually called a block entry or simply an entry. The amount of space that an entry occupies in a block varies directly with the amount of data being stored.

Nodes that contain numbers between 0 and  $\pm 327.67$  require less space than those that contain numbers between  $\pm 327.68$  and  $\pm 21474836.47$ . Nodes that contain character strings can vary greatly in size since a string can consist of 0 to 132 characters. A node with a 1-character string is about the same size as the node that contains a number between  $-327.67$  and  $+327.67$ . However, a node with a string of six characters would require twice as much space.

The total number of entries that a block can hold also varies as a result of this.



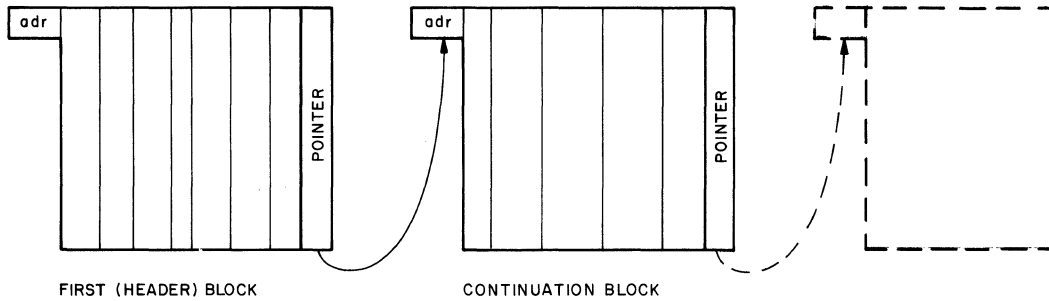
*Some blocks may have many small entries.*



11-2007

*Other blocks may have a few large entries.*

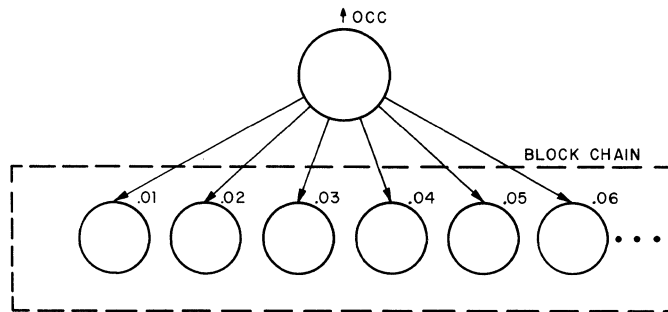
When an entry is too large to fit into the space remaining in a block, MUMPS puts it in an empty block. This block is called a *continuation* block. It's called continuation block because it is used to continue the logical sequence of data that began in the previous block. MUMPS logically connects these blocks by linking them together. The address of a continuation block is inserted into the last entry space of the preceding block. This entry is often called the *continuation block pointer* since it *points* to the continuation block. MUMPS provides as many continuation blocks as are necessary to contain the entries.



11-2008

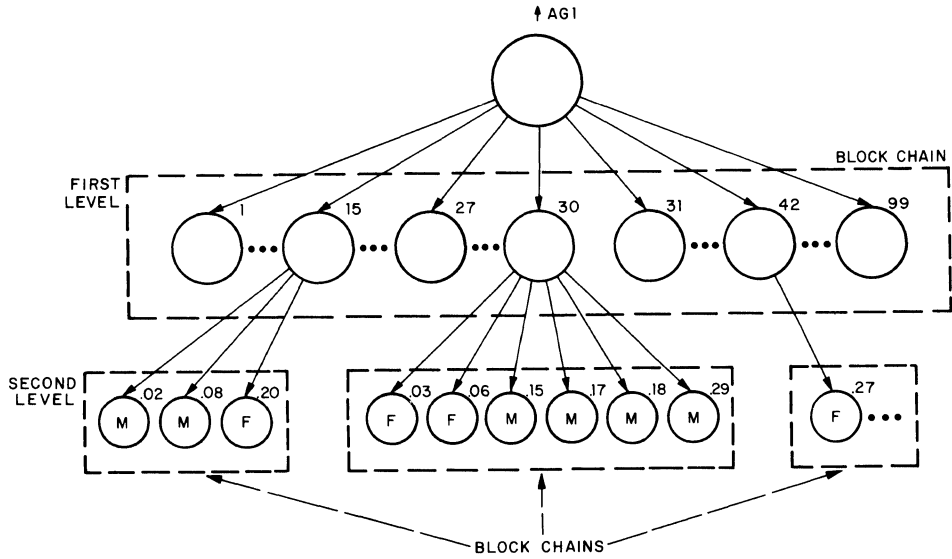
This header block/continuation block relationship is the fundamental structure used in global data storage. Blocks linked in this way are often referred to as a chain. Each logical section of an array resides in a chain.

For example, our  $\uparrow$ OCC array has one subscripting level.



11-1983

All nodes in this array reside in the same chain. However, in an array like  $\uparrow$ AG1, the nodes reside in various chains, depending on the subscript level.



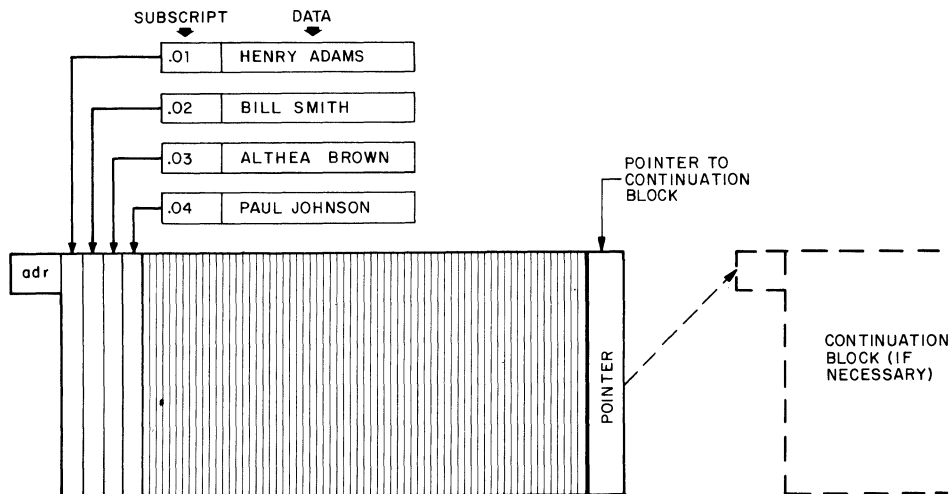
11-1984

*Notice that nodes having a common first subscript reside in the same chain.*

In addition to the numeric or string data, a node also contains a subscript so that it can be located within its block. This is the same subscript that your program uses when referencing the node.

Let's take a close-up look at the entries in some global data blocks. We'll take one from the ↑ AGE array and one from the ↑ NAM array.

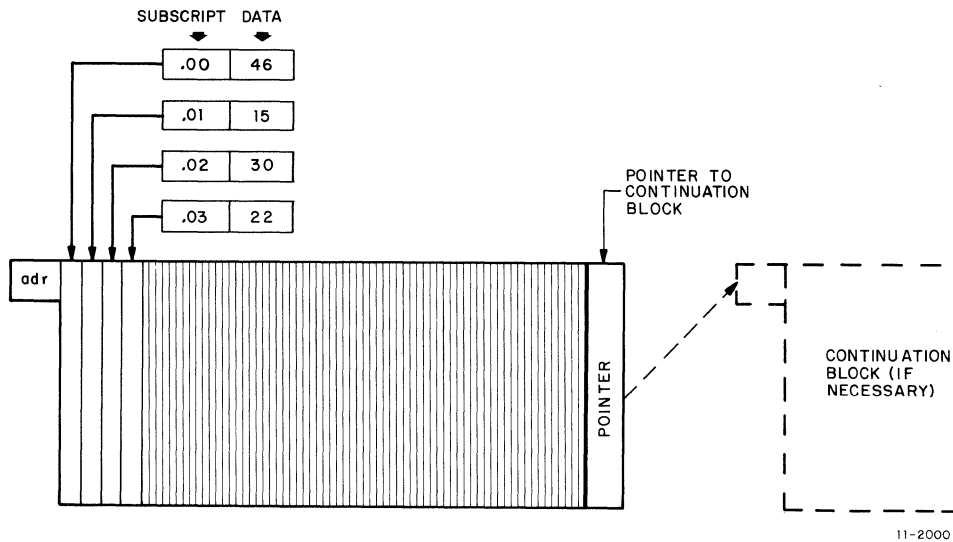
First the ↑ NAM array.



11-1999



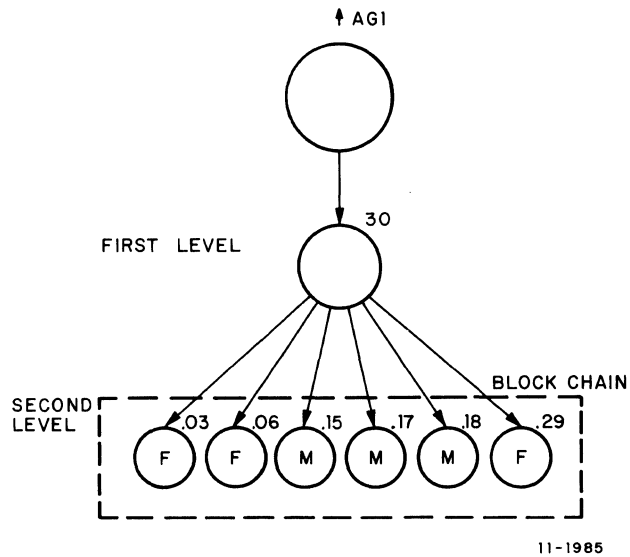
Then the ↑AGE array.



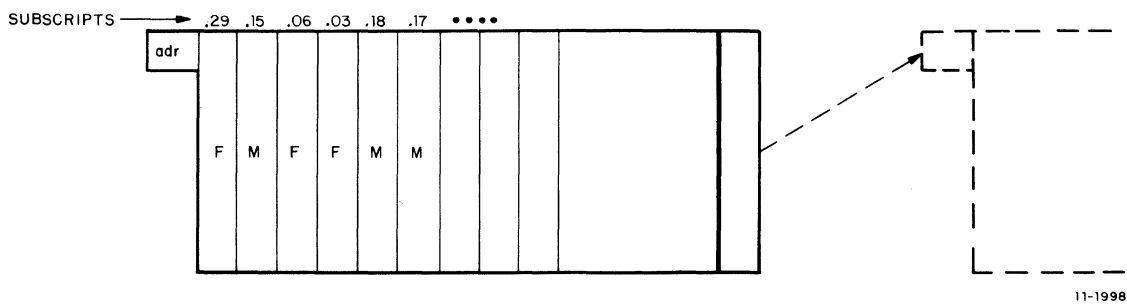
Notice that in both arrays the entries are in ascending subscript order. Why is this? MUMPS simply created these globals the way you asked it to. It stored the global data in the disk blocks in chronological order, regardless of subscript value. Our input programs assigned the lowest subscript to the first name and age to be input -- that's why the data is in subscript order.

It's important to understand that the *physical* and *logical* arrangement of global data need not coincide.

It's not uncommon for the global entries in a block or chain of blocks to be completely out of subscript order. A node is simply placed in the next available space in the block. We make this distinction in our diagrams by showing *logical* global structures (nodes) as a series of circles interconnected by straight lines. Here, the nodes are always shown in left-to-right ascending subscript order.



We show the *physical* structure of a global by using segmented boxes to represent the global data blocks on the disk. Here, global data is shown in the order in which it actually resides in a block.

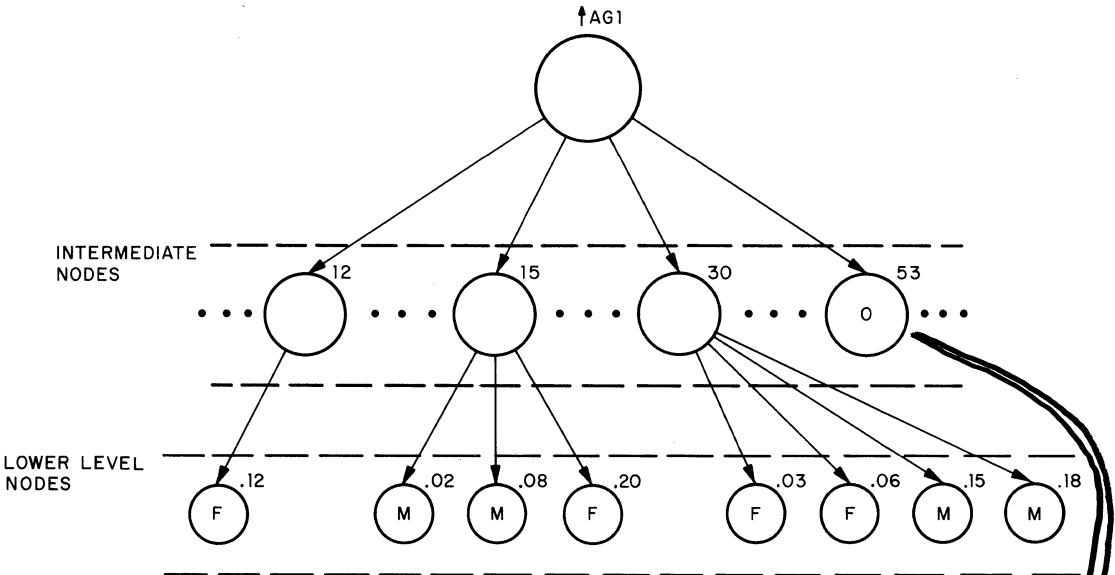


You and your programs must decide how the data is to be arranged and accessed. There are two functions to let you do this: \$HIGH, which you already know, and \$QUERY, which you do not know. \$Q is similar to \$H except that it deals with global variables at the *physical* level.

Nodes can be accessed in *logical* (ascending subscript) order using the \$HIGH function. MUMPS searches the block chain for the next higher subscript from that specified.

Nodes can be accessed in *physical order* using the \$QUERY function (covered in the MUMPS-11 Language Manual). MUMPS simply obtains the next physically higher entry in the block chain.

When you create a global that has more than one level (i.e., subscript), MUMPS places an additional piece of information in certain nodes. These are called intermediate nodes, that is, nodes that provide a path or that point to other nodes at a lower level.



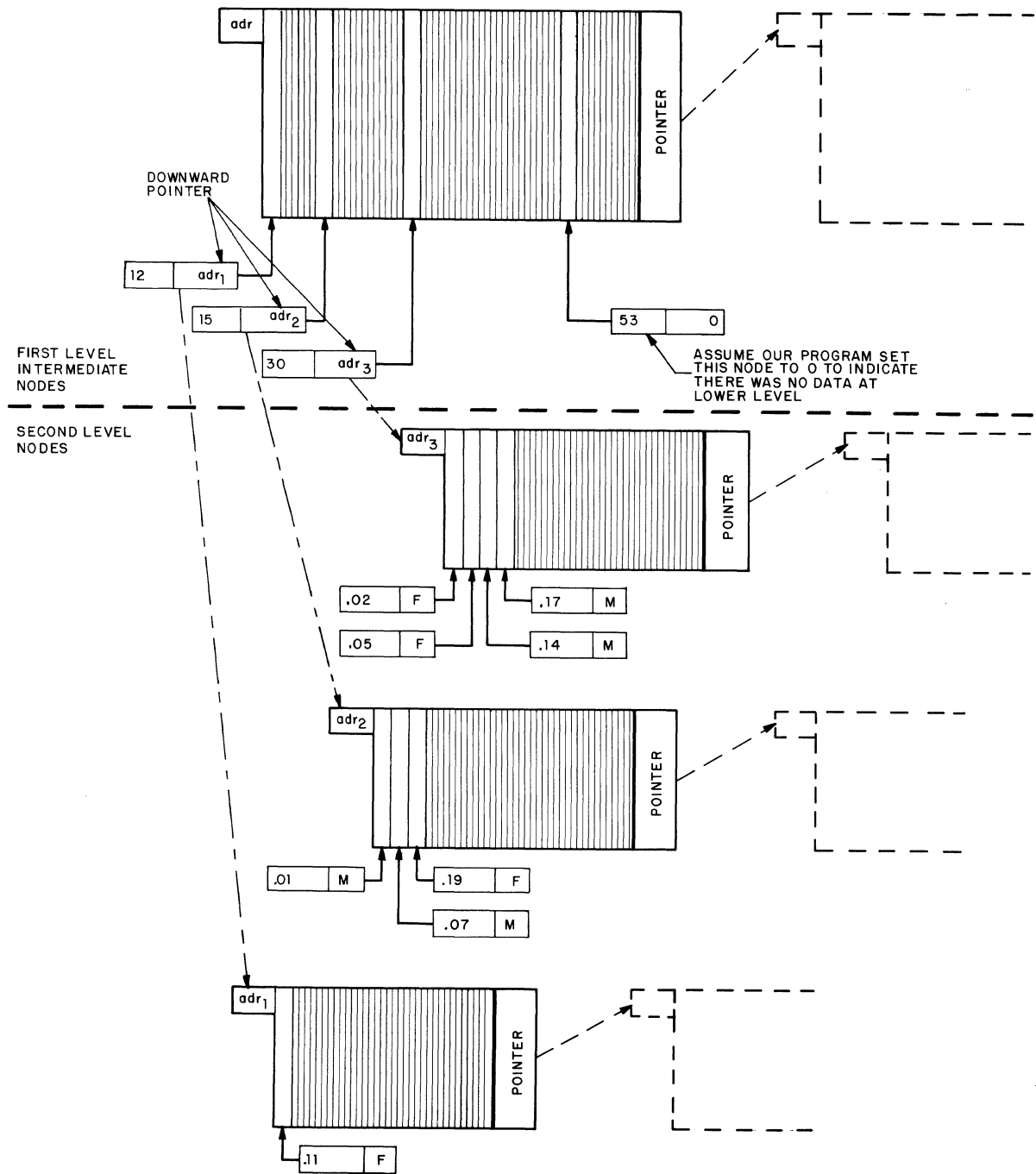
11-1986

The intermediate nodes 12, 15, and 30 point to the nodes at the lower level.

Since there are no lower level nodes, this node has no pointer -- it is not an intermediate node.

The information MUMPS puts into an intermediate node is called a *downward pointer*. Like the pointer that locates a continuation block, a downward pointer is a block address -- the address of the header (i.e., first) block of a chain that contains the nodes at the next lower subscript level.

Here's the physical arrangement of the portion of ↑ AG1 shown above.



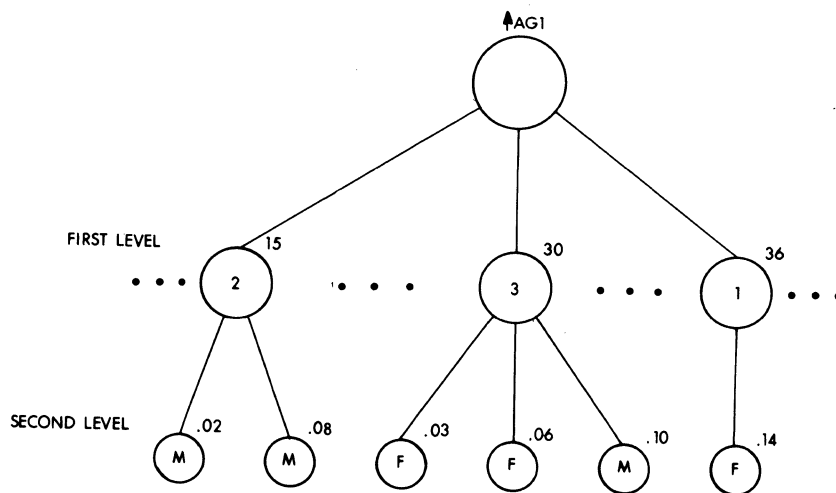
Now you know all the data elements that can reside in a node. In addition to its subscript, a node can contain:

- Either numeric or string data
- A pointer to the next lower subscripting level

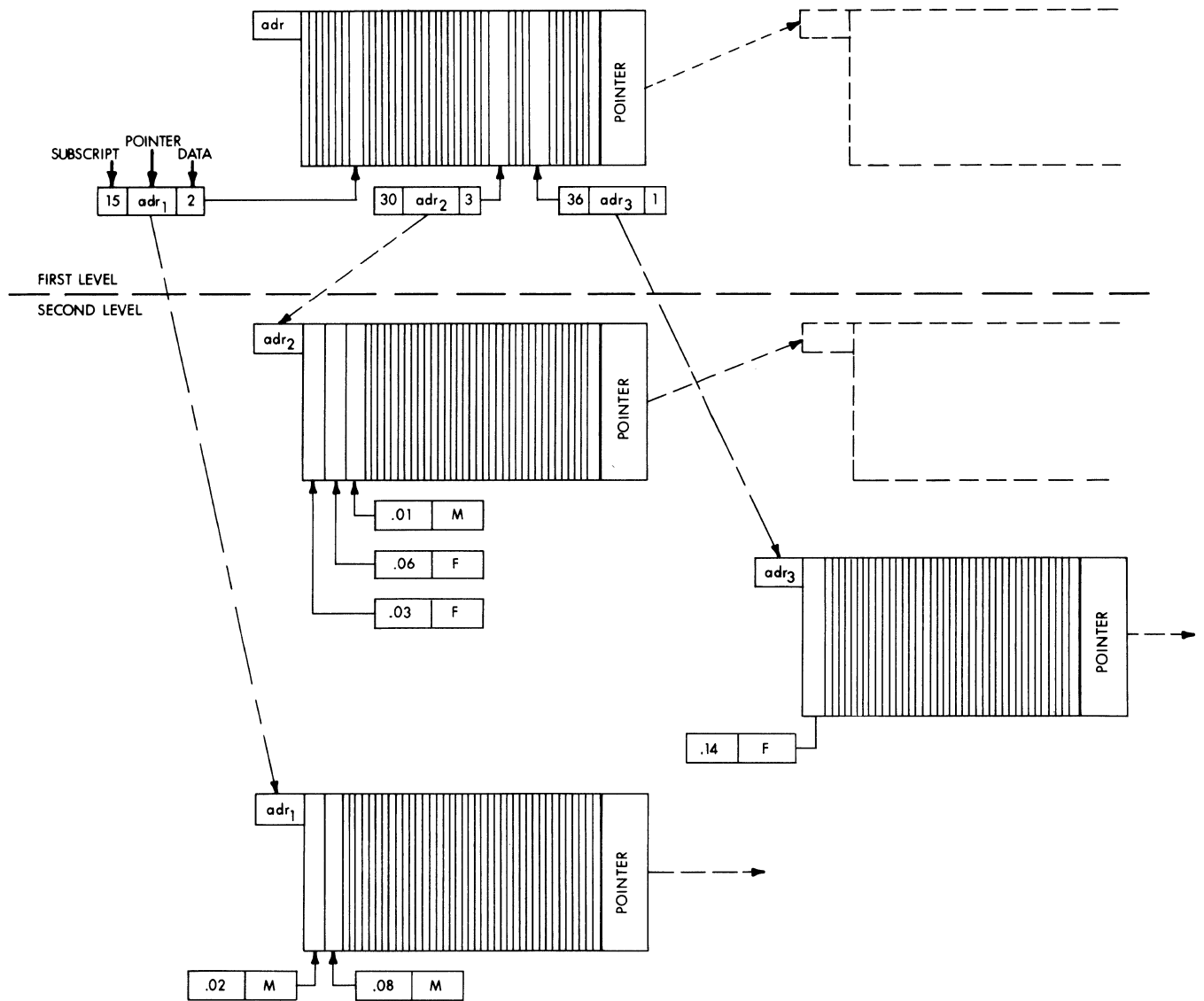
A node can also contain *both* numeric data and a pointer *or*, string data and a pointer. We didn't show you an example above because a node of this type is not part of our data base.

But, we could have structured our data base so that the nodes at the first level did contain data, for example, the number of current second level entries (thereby eliminating the need for counting each time a total is required). Here's what it might look like:

First, the logical layout.

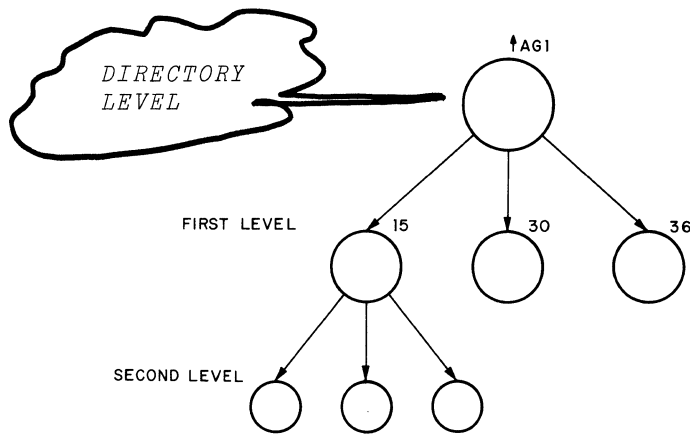


Now, the physical layout.



## ONE MORE THING

There's just one more thing you must know to complete your understanding of global data structures -- how MUMPS knows where the disk blocks that make up your globals begin. We've already shown you that each logical subscripting level has a physical counterpart as one or more blocks. There's one additional level which is higher than any subscript level -- it's called the *directory level*.

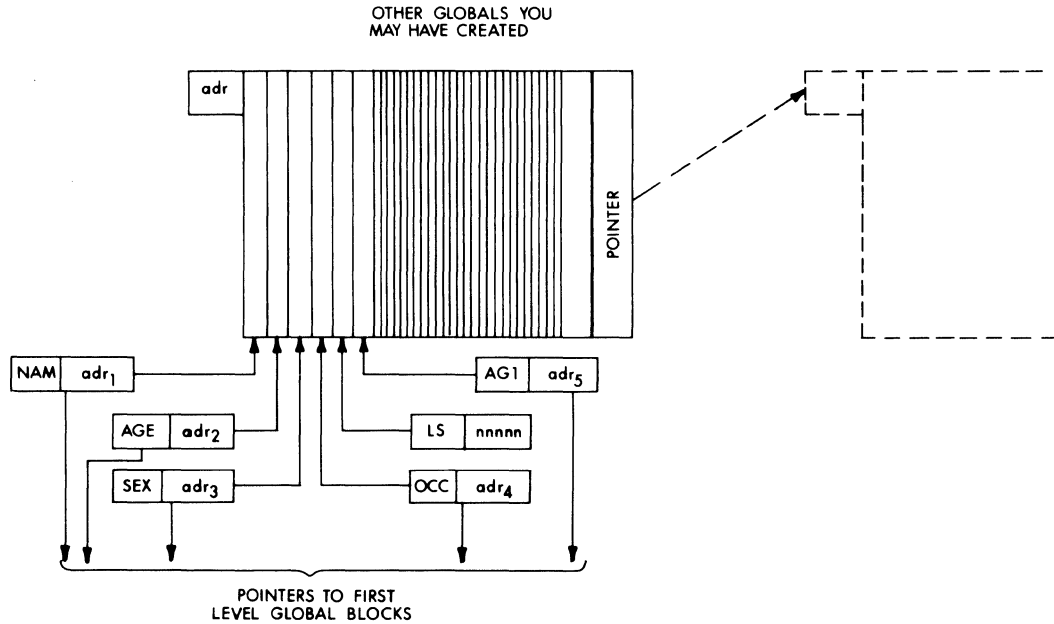


11-1994

All the globals you may create originate in a special block or chain of blocks called the *global directory*. The global directory is similar to other global block chains. It contains a number of entries -- one for each global you create. If there are more entries than can fit in one block, a continuation block is added to form a chain. Each global entry is placed in the directory in the order in which it was created.

Each directory entry is identical in form to the entries in other global blocks. The "subscript" in this case is the global's name. This lets MUMPS locate any global that you or your programs may wish to access. The directory entry can also contain a downward pointer which is the address of the header block of the first subscript level. Like other nodes, a directory entry can also contain string or numeric data.

Here's what your global directory looks like:

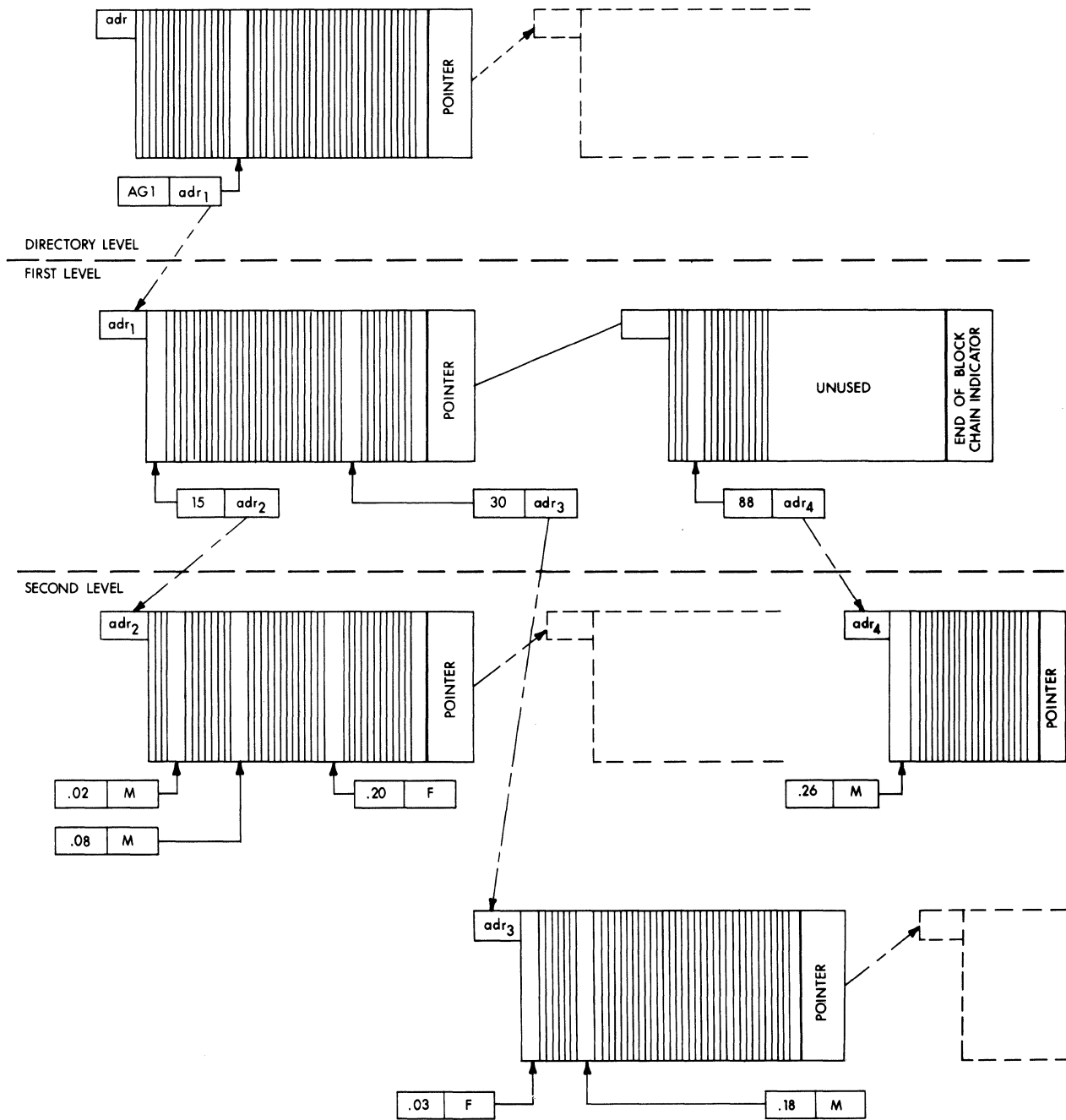


#### NOTICE

MUMPS doesn't store the up arrow (↑) that precedes the global name, since its only function is to tell MUMPS which of your program's variables are local and which are global.

Now you know the whole story! How MUMPS *really* stores global data. Let's put all the pieces together -- look at the structure of our AG1 global. We haven't space to diagram all the blocks your data may have used so we'll show just a few of the header blocks so you'll get the idea.





Let's summarize some of the important points about global data structure:

- A global consists of one or more disk blocks. When a global consists of more than one block, pointers are used to link the blocks together. Pointers are block addresses -- they tell MUMPS which disk blocks belong to your global(s). There are two types of pointers, continuation block pointers and downward pointers. Downward pointers point to a block that contains nodes at the next lower subscript level.
- All globals originate in your global directory. Each directory entry consists of a global name as well as a block pointer and/or data (either string or numeric).
- Every global has at least one level -- the directory level. Other lower levels are created by the use of one or more subscripts. Each additional subscript creates an additional lower level.

## HOW MUMPS REALLY ACCESSES GLOBALS

Back in the beginning of the chapter, we told you how both the full and naked global references *appear* to work. Now we can tell you how they really work, since you now know the physical structure of global data. If you'll remember, we told you that whenever MUMPS obtains or places global data on the disk, it reads or writes a block. It reads this block into main memory so it can find the global information your program requested. If you have a program that makes lots of global references, chances are MUMPS is going to be reading and/or writing lots of disk blocks. That can take a lot of time, particularly if you don't use naked references whenever possible. Time is the enemy of the MUMPS programmer. Whenever he writes programs that waste time, it's not only his time -- it's everyone's time. It's the time that the user of his program wastes. It's the time wasted by other users and their programs while waiting for their turn to use the disk. Remember, MUMPS *is* a timesharing system.

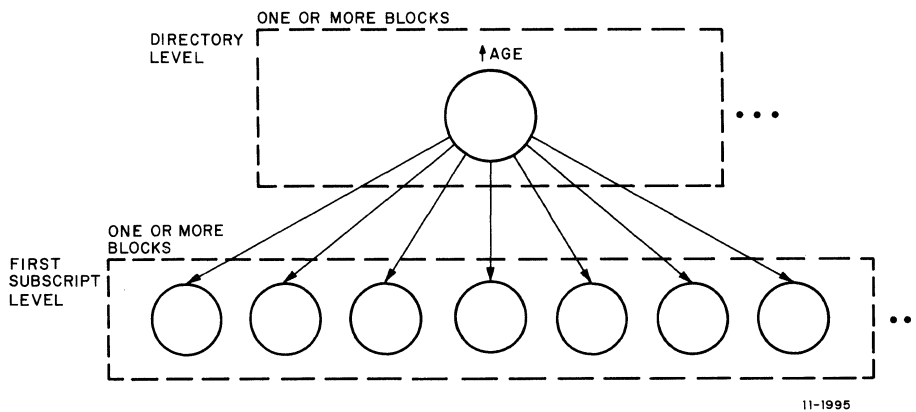
MUMPS accesses global data in one of two ways, depending on the type of global reference used: full reference or naked reference.

Let's begin with the full global reference. Every logical level in a global, including the directory, occupies at least one block, even if there is only one entry in each block. Additional continuation blocks are added as space is required.

When MUMPS encounters full global reference, it must always read at least one block from the disk for each logical level. When levels have continuation blocks, as many of those blocks must also be read in as are required to locate the specified node entry.

For example, when we used the A1 program to search the ↑AGE global, MUMPS had to read at least two blocks each time a global reference was made, one block for the directory and one block for the data at the first level.

Here's what ↑AGE looked like.



And here's step 3.2 -- the step that contained all the global references.

```
3.20 S I=$H(↑AGE(I)) I I=>0, ↑AGE(I)=X S TOT=TOT+1
```

*At least two blocks read here*

*At least two blocks read here.*

So, each time step 3.2 is executed, at least four blocks must be read into memory from the disk. If there are continuation blocks at either the directory level or the first level, more blocks might have to be read in.

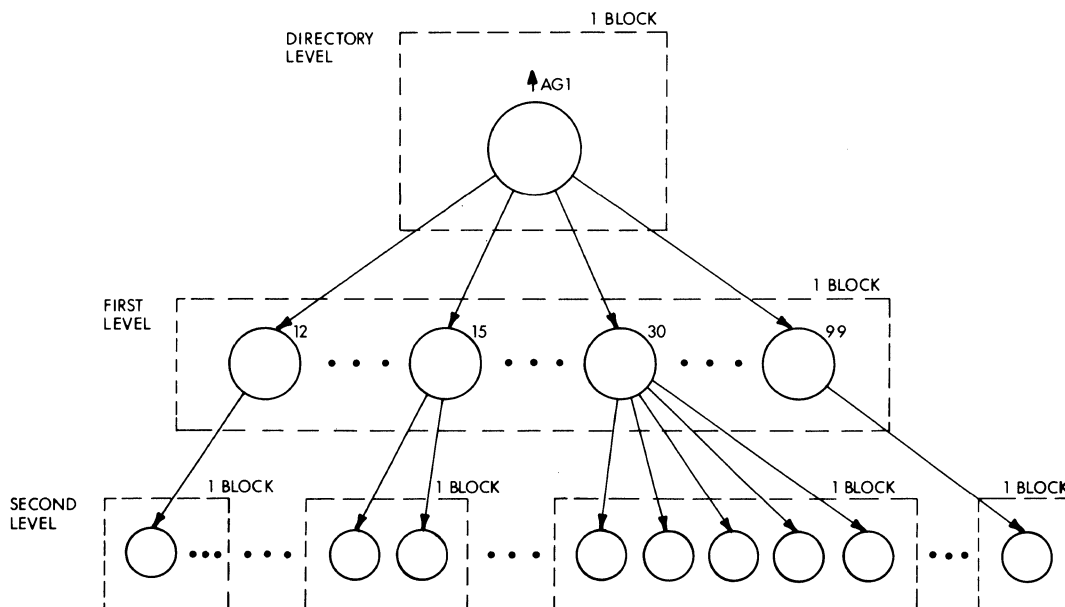
- If you have a large number of globals, the global's name could be located in the third continuation block of the directory. Then, each time a full reference is made, four blocks would have to be read simply to find the name.
- If you have more than a small amount of data, some of it would have to be read-in from continuation blocks at the first level.

Let's say AGE contains only 30 entries -- they'd probably all fit in one block. Let's also assume there are only five age 30 entries and that your directory is only one block long. How many blocks (in other words - disk accesses) would be required to search all the entries for people age 30? That's right -- 122 accesses!

*For each full global reference, the directory block is read into memory 30 times then the first level block is read into memory 30 times. This is done for each entry. This makes a total of 160 disk accesses. The two additional accesses occur when the search fails to find any higher nodes. The time required for such a simple operation is significant.*

Let's try another example. This time we'll see how many blocks are used when the A2 program accesses the AG1 global.

AG1 looked something like this, remember?

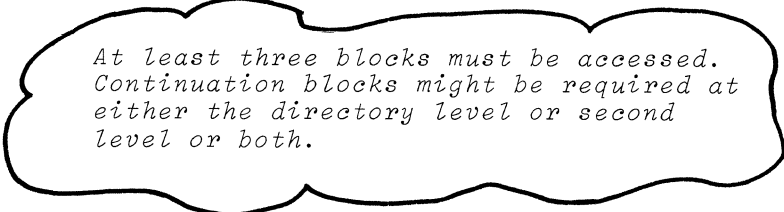


In AG1, at least three blocks must be read to access a node at the lowest level.

- At least one block for the directory
- One block for first-level nodes (representing ages 1 through 99)
- At least one block for each node at the second level

Here's the step in A2 that performed the global access:

```
3.20 S I=$H(↑AG1(X,I)) I I=>0 S TOT=TOT+1 G 3.2
```



*At least three blocks must be accessed. Continuation blocks might be required at either the directory level or second level or both.*

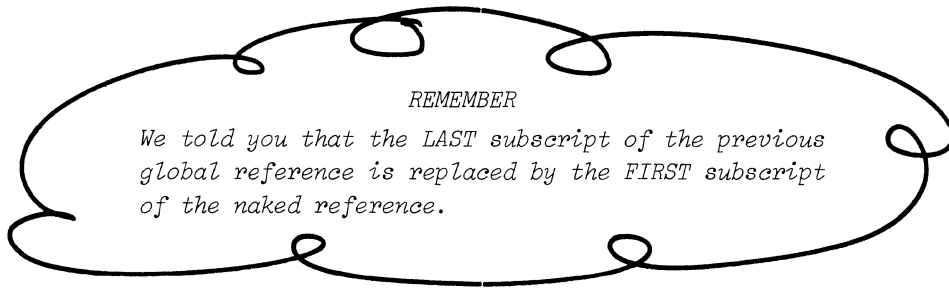
The way this global is designed, all similar age entries reside at the second level in the same block or a continuation block. This time every node does not have to be examined, as with ↑AGE, to find all members of a particular group.

Now to our test! Again, we'll assume that there are only 30 entries in the global, 5 of which are in the age 30 group. How many blocks will now be accessed to obtain the total number of age 30 entries? Right! 18 blocks -- a lot less than the 122 blocks in the previous example. This is better, but not the best we can do. There's another way ...

#### THE OTHER WAY

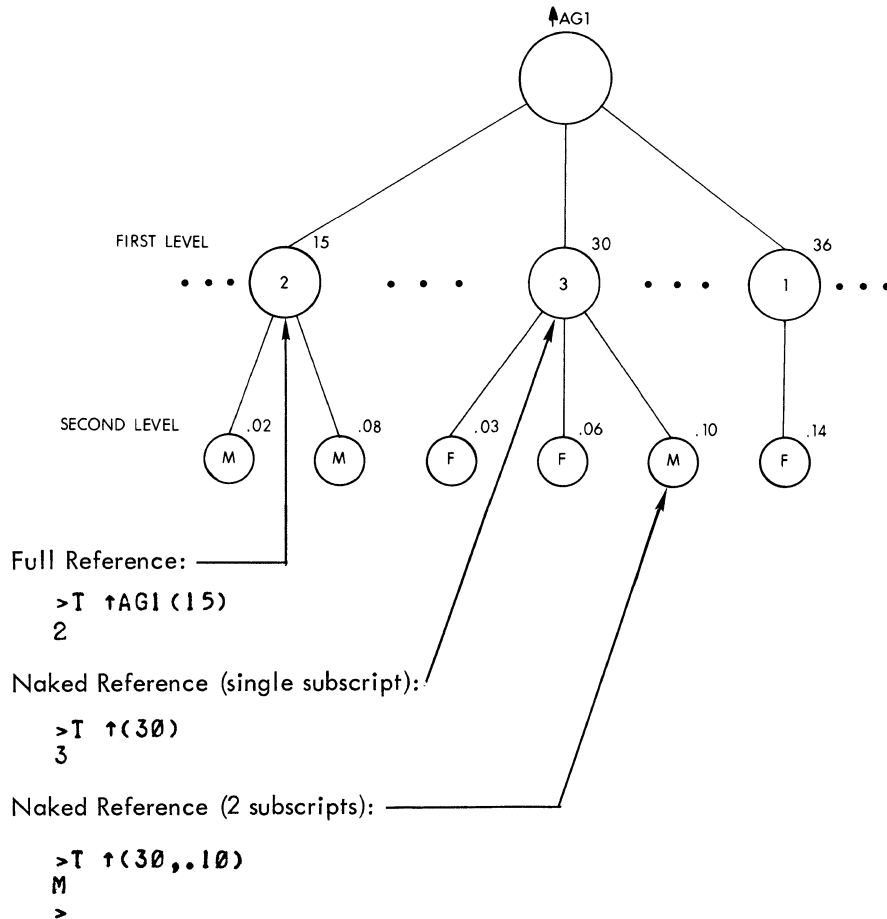
The other way MUMPS accesses globals is through use of the naked reference. As we told you earlier, MUMPS "remembers" some information about each global access it's requested to perform. Each time an access is completed, this information is updated. When a full global reference is requested, MUMPS doesn't use this information -- it makes the access via the global directory each time and down the tree to the specified node. But, when a naked reference is requested, MUMPS uses the information it "remembered" about the last access. It uses this information by allowing the current naked reference *to begin at the level reached by the previous global reference*. This reduces the number of blocks that need to be accessed to find the requested node each time.

For example, if the last reference was at the first subscript level, then a subsequent single subscript naked reference would also be at the first level.



A two-subscript naked reference would also start at the second level, but when the node specified by the first subscript was found, the access path would continue down to the third level node specified by the second subscript.

For example:



The information that MUMPS uses to perform a naked reference is:

- the address of the first block of the level reached by the previous global reference (called GHEAD)
- the address of the block that was currently in memory at the time of the previous global reference (called GBLOCK)

Since this is the only information MUMPS uses, a naked reference can't be used to access a level that is higher than established by the previous access. Because with naked access MUMPS *knows* only the address of first block of the last level reached and the address of the last block accessed at that level. It doesn't *know* the name of your global or how it reached the current level. MUMPS' globals are linked together with the downward pointers and continuation pointers; there are no upward pointers.

When MUMPS performs a naked reference, it does the following:

- First it checks to see that the user's GBLOCK is in memory.<sup>1</sup>
- If GBLOCK isn't in memory, it reads in the GHEAD block (at that level), then continues accessing as many blocks as required to reach the node at the specified subscripting level.
- If GBLOCK is in memory, MUMPS then checks to see whether the first subscript specified in the naked reference is in this block. If it isn't, GHEAD block is read in, and the search is made again. If it is not found, continuation blocks are read in, if they exist. When the node is found, as many other blocks are accessed as are necessary to reach the node at the level of the last subscript.
- If the first subscript is in GBLOCK, MUMPS accesses the node. If there aren't any additional subscripts in the naked reference, the access is complete. Otherwise, as many blocks as are required are accessed to reach the node at the specified subscript level.

---

<sup>1</sup> Since MUMPS is a timesharing system, other users may have used the disk. Your GBLOCK may not be in memory when MUMPS gives you your time slice.

In terms of number of blocks accessed:

Best case - no blocks are accessed:

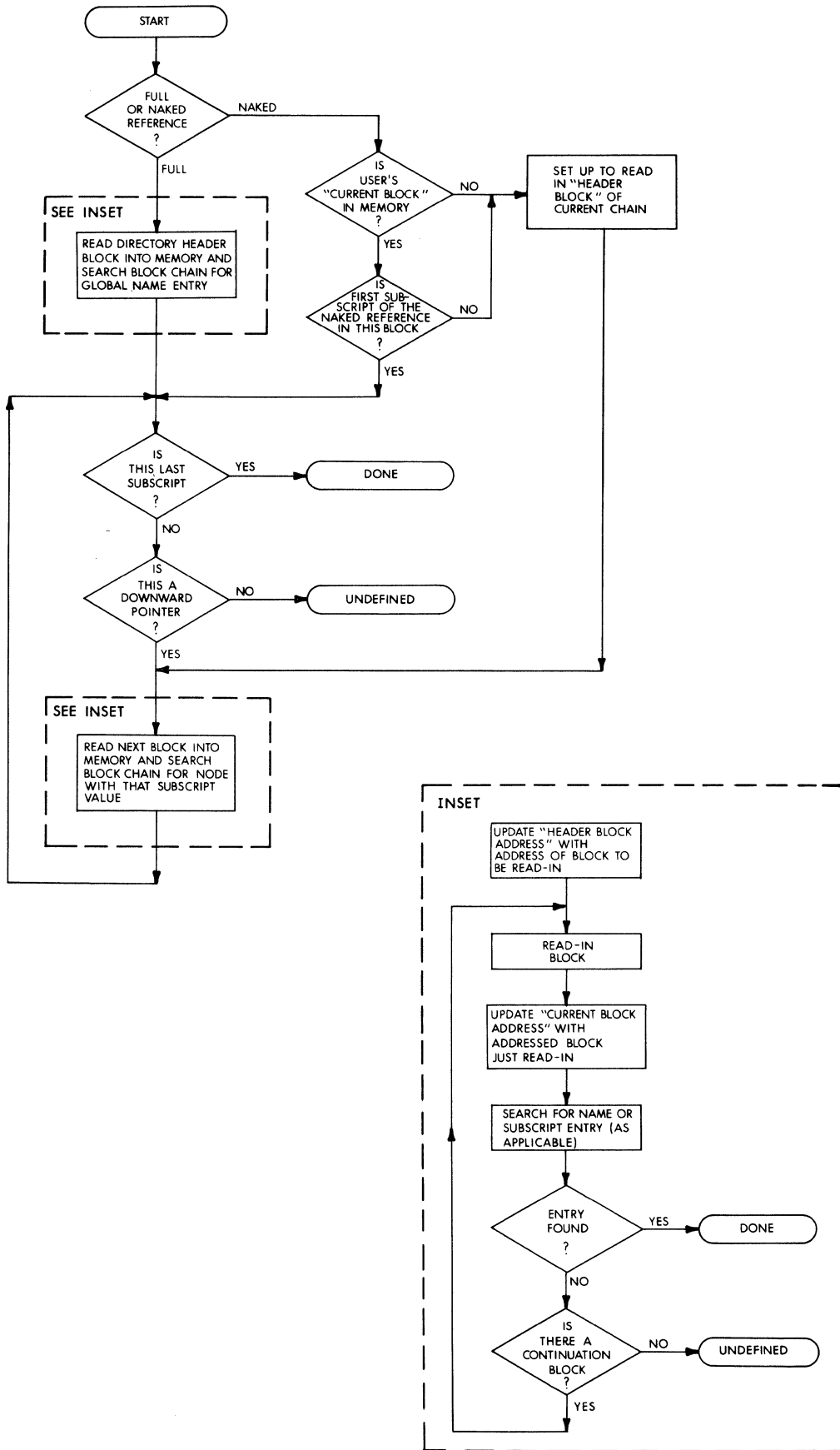
- Your GBLOCK was in memory  
and
- The first subscript in your naked reference was in GBLOCK  
and
- There weren't any additional subscripts and therefore no other  
lower level blocks to access.

Worst case - At least one block for each subscript level in naked reference is accessed.

- Your GBLOCK wasn't in memory (GHEAD block had to be read in)
- The first subscript of your naked reference wasn't in GHEAD block  
(a continuation block had to be read in)
- Your naked reference specified more than one subscript level, thus  
at least one block for each additional subscript level had to be read in  
to locate the specified node.

With what we've told you so far, the following flowchart of how MUMPS accesses global data should clarify any questions you may have.





OK, let's go back to the ↑AGE and ↑AG1 globals. This time, we'll see how many blocks have to be read in when naked references are used.

We'll begin with ↑AGE. Here's part 3 that contains the naked references in the A3 program.

```
3.10 S TOT=0,I=$H(↑AGE(-.01)) I I<0 Q
3.15 I ↑(I)=X S TOT=TOT+1
3.20 S I=$H(↑(I)) I I=>0,↑(I)=X S TOT=TOT+1 G 3.2
3.30 I I=>0 G 3.2
```

As mentioned earlier, we'll assume the directory is contained in one block and the 30 age entries are in one block.

How many blocks do you think will be accessed now? Two? Right -- for the best case! The full global reference provides the starting point for the naked reference that follows. The two blocks are accessed here -- one for the directory and one for the first subscript level. All other global references are naked and occur at the same subscript level.

*NOTE*

*If other MUMPS users are also waiting to use the disk, additional blocks may have to be accessed since your GBLOCK might not be in core when MUMPS returns to you after servicing another user.*

Now let's see how many blocks of the ↑AG1 global are accessed when the A4 program runs.

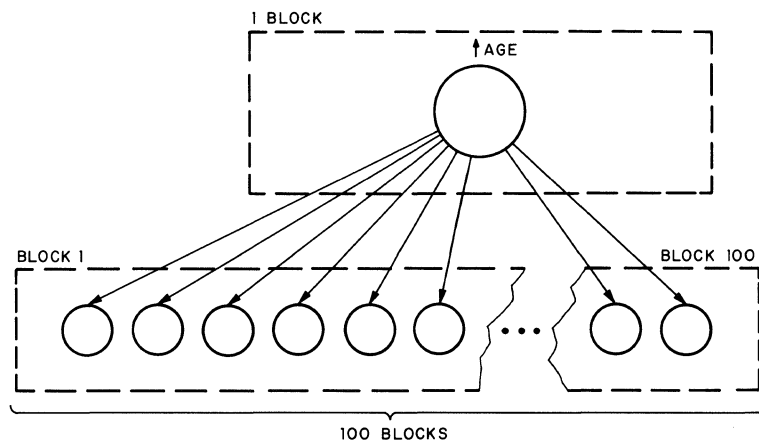
Here's part 3 of A4:

```
3.10 S TOT=0,I=$H(↑AG1(X,-.01)) I I<0 Q
3.15 S TOT=1
3.20 S I=$H(↑(I)) I I=>0 S TOT=TOT+1 G 3.2
```

As we discovered earlier, a full global reference to the second level of ↑AG1 requires that at least three blocks be accessed. No additional blocks need be accessed thereafter because our five age 30 entries reside in the same block. That block is already in memory as a result of the full global reference.

At this point you may be thinking that there is no advantage to using the ↑AG1 global once naked references are incorporated into our programs. In this particular instance, you're right! Under the test conditions we've specified, naked reference used with the ↑AGE global works faster. Only two (or possibly three) blocks need be accessed. However, if these globals contained a census data base of realistic size, this would not be true.

For example, to get a more reasonable view of the situation, suppose that the census data occupied 100 blocks in the ↑AGE global, and that 7% of the population was age 30.

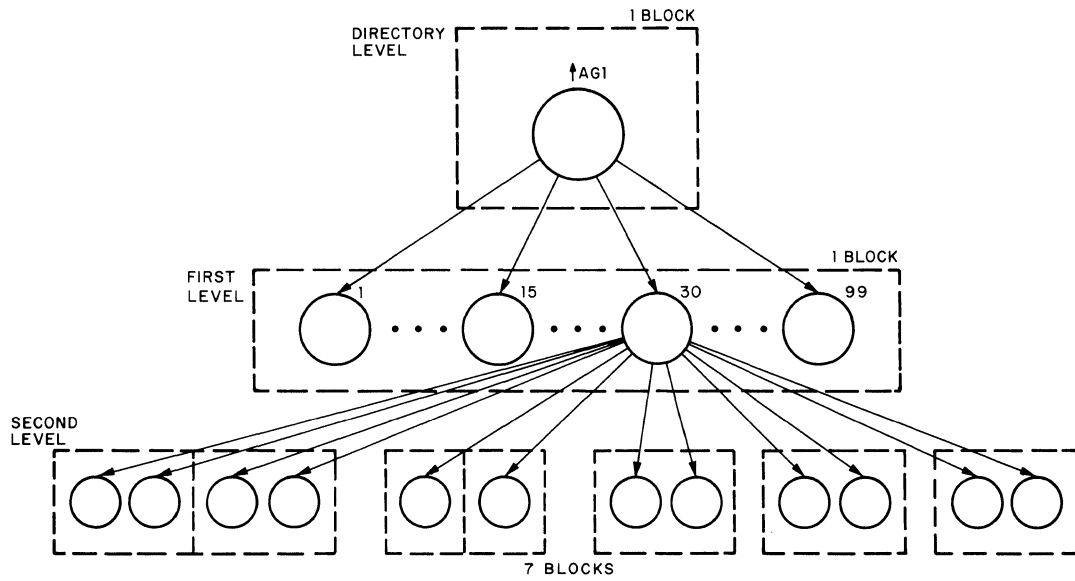


11-1996

Now how many blocks must be accessed to get the total number of age 30 entries? 101 blocks!

- One block for the directory (our original assumption)
- The first block in the chain (GHEAD)
- Ninety-nine continuation blocks.

Try the same test with ↑AG1. Here's what ↑AG1 might look like (we'll show only the age 30 nodes to save space):



11-1997

How many blocks are accessed now?

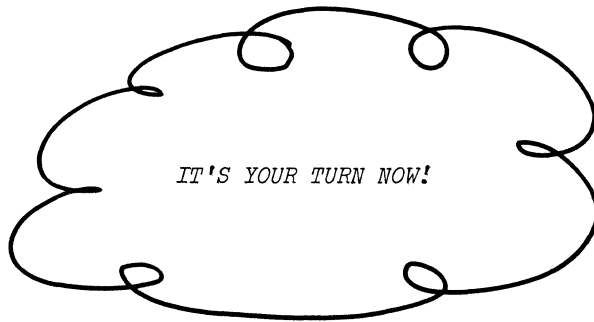
1 for the directory	1
1 for the first level	1
1 for the second level	1
6 continuation blocks	<u>6</u>
	9 blocks

*NINE BLOCKS IS A LOT LESS  
THAN 101 BLOCKS!*

## BECOMING FLUENT

No more chapters left! This is the end of our story. Now that you know a little about MUMPS-11, where do you go from here? As we told you in the Preface, this is only an introduction to MUMPS-11. We've just tried to point you in the right direction. Bear up! Help is on the way. It's time to read the MUMPS-11 reference manuals described in the preface. You can get them from your trustworthy computer supplier, Digital Equipment Corporation. When ordering, be sure to use the order number as well as the name of the manual. Begin with the *MUMPS-11 Language Reference Manual*; then progress to the *Programmer's Guide* and the *Operator's Guide*.

Continue reading and studying.



## GLOSSARY OF TERMS

Array	An array is a group of subscripted variables that have a common identifier. An array can consist of either local or global variables.
Binary Operator	A binary operator is an operator that requires two operands (expression elements).
Boolean Valued Expression	A Boolean Valued Expression (bve) is an expression, which, when evaluated, produces either a True (-0.01) or False (0) result.
Command	A command is the principal algorithmic component of the MUMPS Language. MUMPS commands consist of a set of keywords that characterize actions. (e.g., GOTO, SET HALT, RUN, etc.)
Concatenation	Concatenation is the process of linking together two or more string data elements to form a single string. Concatenation is a string expression operation that is designated by the commercial 'at' sign (@).
Constant	A constant is a quantity within the range of legal MUMPS numbers ( $\pm 21474836.47$ ) explicitly stated in an argument to a command or as an operand in an expression.
Data Base	Data base is that body of disk-stored information residing in global arrays.
Direct Mode	Direct Mode is the mode of system operation that enables the programmer to: <ul style="list-style-type: none"><li>a. enter commands for immediate execution</li><li>b. create or modify a program.</li></ul>
Directory	A directory is a disk-resident table that contains the names and disk starting address of either programs or global files. Each User Class Identifier in a MUMPS-11 system has two directories associated with it: a program directory, and a global directory.

Expression	An expression is any legal combination of operands (elements) and operators. Legal expression elements include: literals, constants, variables, subexpressions, and function references. An expression may consist of a single element, an element/operator combination or a series of element/operator combinations.
Expression Element	An expression element is the operand component of a MUMPS expression.  An expression element may be: a constant, a simple variable, a literal, a local subscripted variable, a global variable, a function reference or a subexpression.
Function	A function is a MUMPS expression component that invokes an algorithm the result of which is an expression element (operand). Each MUMPS function is assigned a unique mnemonic, the first character of which is the dollar sign (\$) symbol.
Global	A global is a tree-structured data file stored in the common data base on the disk. Globals comprise an external system of symbolically referenced arrays.
Global Variable	A global variable is a subscripted variable that forms an element (or node) of a global array.
Identifier	An identifier is a name consisting of one to three alphanumeric characters. The first character must be either an alphabetic character or the percent (%) symbol. Identifiers are used as names for variables, programs, and globals. The percent symbol must be used as the first character of a Library Program or Global name.
Indirect Mode	Indirect Mode is that mode of system operation in which the steps of a stored program are executed. In this mode of operation, commands cannot be entered from the terminal and programs cannot be created or modified.
Indirect Reference	An indirect reference is a feature of the language that permits a string variable to represent a command's argument or argument list. In operation, the string value of the variable is taken as the argument or argument list. The indirection symbol, back arrow (←) or underscore (_), must precede the variable reference.

Literal	<p>A literal is the explicit representation of character strings in expressions and in command and function arguments by delimiting them with quotation marks (" "). Literals may not contain:</p> <table> <tr> <td>quotation marks</td> <td>CTRL C</td> <td>ALTMODE</td> </tr> <tr> <td>Carriage RETURN</td> <td>CTRL O</td> <td>Vertical Tab</td> </tr> <tr> <td>FORM Feed</td> <td>CTRL U</td> <td>RUBOUT (DEL)</td> </tr> <tr> <td>LINE FEED</td> <td>NUL</td> <td></td> </tr> </table>	quotation marks	CTRL C	ALTMODE	Carriage RETURN	CTRL O	Vertical Tab	FORM Feed	CTRL U	RUBOUT (DEL)	LINE FEED	NUL	
quotation marks	CTRL C	ALTMODE											
Carriage RETURN	CTRL O	Vertical Tab											
FORM Feed	CTRL U	RUBOUT (DEL)											
LINE FEED	NUL												
Local Variable	<p>A local variable is a variable that resides in the partition of the program that created it (as opposed to a global variable).</p>												
Naked Reference	<p>The naked reference is a feature that provides an abbreviated method for accessing global variables to reduce disk access time. This permits subsequent references to a global to be made simply by specifying an up-arrow (↑) followed by one or more subscripts. The variable name is assumed from the last global reference in which a name was explicitly stated. The first subscript in the naked reference replaces last subscript in the previous reference (either naked or complete). Using the naked reference reduces disk access time since the search for the specified node begins at the subscripting level attained by the last global reference rather than at the global directory level.</p>												
Node	<p>A node is a global array element addressed by a subscript.</p>												
Numbers	<p>Numbers in MUMPS are signed fixed-point quantities in the range <math>\pm 21474836.47</math>. Decimal fractions greater than two places are truncated to two places.</p>												
Numeric Valued Expression	<p>A numeric valued expression (nve) is an expression which, when evaluated, produces a numeric result.</p>												
Operator	<p>An operator is a component of a MUMPS expression that invokes an algorithm to perform either arithmetic, string, or Boolean manipulations. (See binary operator and unary operator.)</p>												
Part Number	<p>A part number is the integer portion of a step number and is used to refer collectively to all steps having a common integer base.</p>												
Partition	<p>A partition is the memory area within which a job resides. A partition is allocated to a job either at terminal log-in time or upon execution of the START command. A partition contains both program and local variable storage areas as well as program state information necessary for timesharing operation.</p>												



Pattern Verification	Pattern verification is a feature that permits evaluation of text strings for the occurrence of desired combinations of alphabetic, numeric and punctuation characters. Pattern verification is specified by the "?" operator followed by Pattern Specification Codes (psc).
Program Name	A program name is an identifier that is associated with a particular program. System Library program names must use the percent symbol (%) as the first character.
Programmer Access Code	The Programmer Access Code (PAC) is a three-character code created at System Generation time that allows the terminal user to enter Direct Mode.
Step Number	A step number is a number used to identify each line of a MUMPS program. A step number must be in the range 0.01 - 327.6 and excludes all numbers in this range that are integers.
String	Any contiguous group of up to 132 ASCII characters.
String Concatenation	See Concatenation.
String Valued Expression	A string valued expression (sve) is an expression that produces a string result upon evaluation.
Subexpression	A subexpression is an expression element that consists of any legitimate expression enclosed in parentheses.
Subscripts	A subscript is a numeric valued expression or expression element appended to a local or global variable name to uniquely identify specific elements of an array. Subscripts are enclosed in parentheses. Multiple subscripts must be separated by commas and can be used in global references only.
Subscripted Variable	A subscripted variable is a variable to which a subscript is affixed (see subscript and variable). Both global and local variables are forms of subscripted variables.
System Variable	A System Variable is a variable that is permanently defined within the operating system. These variables provide system and control information to all programs. The first character of a System Variable is always a dollar sign (\$). System Variables are maintained and modified by the operating system and/or system manager only.
Unary Operator	A unary operator is an operator that requires a single operand (expression element).

User Class Identifier (UCI)

A UCI is a three-character code used at terminal log-in time to permit access to the group of programs and global files with which it is associated. When used with the Programmer Access Code, the UCI allows these programs to be modified and new programs to be created.

Variable

A variable is the symbolic representation of a logical storage location. Specific types include: local, global, simple and subscripted variables. Variables are symbolically referenced by means of identifiers.



READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form.

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

---

---

---

---

---

---

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

---

---

---

---

---

---

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

---

---

---

---

---

---

Please indicate the type of user/reader that you most nearly represent.

- Assembly language programmer
- Higher-level language programmer
- Occasional programmer (experienced)
- User with little programming experience
- Student programmer
- Non-programmer interested in computer concepts and capabilities

Name \_\_\_\_\_ Date \_\_\_\_\_

Organization \_\_\_\_\_

Street \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip Code \_\_\_\_\_  
or  
Country

If you require a written reply, please check here.

Please cut along this line.

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**

Software Communications  
P. O. Box F  
Maynard, Massachusetts 01754



**digital**

digital equipment corporation