The User's Guide to North Star BASIC

FIRST EDITION

by ROBERT R. ROGERS

INTERACTIVE COMPUTERS / houston, texas

COPYRIGHT (c), 1978
ROBERT R. ROGERS

Made in the United States of America

Library of Congress Cataloging in Publication Data
Main entry under title:

The User's Guide to North Star BASIC, first edition.

Published and Printed by:

Interactive Computers
7620 Dashwood
Houston, Texas 77036

All rights reserved, including rights of reproduction and use in any form or by any means, including the making of copies by any photo process, or by any electronic or mechanical device, printed or written or oral, or recording for sound or visual reproduction, or for use in any knowledge or retrieval system or device, unless permission in writing is obtained from the copyright proprietor.

preface

About seven months ago a very smart businessman brought to my house a Sol-20* COMPUTER with 24 K memory, a PANASONIC MONITOR, a NORTH STAR MICRO-DISK SYSTEM*, and one VERBATIM MINIDISK with the disk operating system (DOS) and NORTH STAR BASIC - VERSION 6 * RELEASE 3 on it. (I mention all of this detail so that the reader knows exactly what it is that I learned on and used to WRITE the programs included in this book.) He set it up in my library, gave me three little 20 page manuals, plugged it in, had a cup of coffee, and left.

If he had called me and said, "Hey, I've got a great deal for just $3600 --- can I bring it over?" My response would have been very quick and short --- "NO!" I didn't need a $3600 toy.

After he left I went in and pushed the button and thought to myself, "It's a neat television-typewriter." I even picked up one of the manuals, a blue one, read a few pages; but couldn't get the thing to do anything that the book said it was supposed to do. So I picked up another manual, this time a yellow one. I did a couple of the things it said to do and I finally got the "little blue mail box" to come to life. First a little red light came on, then it hummed for a few seconds, then turned off. That's about all I got the thing to do for the first week. I wasn't too impressed.

By the end of the first month, after messing with the thing when I had nothing better to do, I finally got it working to the point that I could copy game programs out of a book; about half of which never worked and I never understood why.

By the end of the third month I was writing a few simple mathematical programs - nothing much - but something.

By the end of the fifth month I wrote what I considered to be a significant program. It was an inventory counting and extension program, which involved a data file. It was a relatively straight-forward program, nothing fancy. It did work and it did save time over the way I had previously done the inventory.

By the end of the sixth month I had several game programs, an invoice preparation program, a product distribution program, a customer mailing list program, a cost accounting program, and a monthly payment program to make computer note payments for the next 36 months to my bank. I am now hooked.

If I had had the information that I have put in this book available to me when I started, in the same "over-simplified" format that's been used, I would have saved hours of real frustration. Instead of six months of "hard labor," I could have discovered what a real "magic box" it is in several weeks.

So you know, that I know you know. I feel a last minute obligation to mention that I have taken certain liberties in some of my explanations and have made absolutes out of some things for which there are many exceptions. For the most part I have knowingly done this, feeling that there is nothing to be gained by confusing the reader.

This is not intended to be a definitive text for the "computer expert." It is a starting point for somebody who has never sat in front of one of these "T.V. looking things" and wants to learn how they work. If you feel offended or have a great urge to tell me about some of my "mistakes," please do. For you who are just starting, consider your finding these "mistakes" a yard stick of your progress.

I hope you, as a potential reader, derive as much pleasure and open a gateway to a whole new world of logic as I have.

GOOD LUCK AND MAY YOUR MEMORY NEVER BE FULL.

R.R. ROGERS

*Sol is a trademark of Processor Technology Corp., Pleasanton, CA
North Star DOS and BASIC are copyrighted products of North Star Computers, Berkeley, CA, and are licensed for use only with the North Star MICRO-DISK SYSTEM.

contents

P R E F A C E
A C K N O W L E D G M E N T
I N T R O D U C T I O N

Chapter 1. How do we get STARTED. 1
Chapter 2. How to WRITE a program 5
Chapter 3. LINE NUMBERS 11
Chapter 4. How to DISPLAY a program 13
Chapter 5. VARIABLES. 17
Chapter 6. Cast of characters 21
Chapter 7. The RENUMBER function. 33
Chapter 8. How to correct or EDIT a BASIC program 37
Chapter 9. How to LOAD a program from a mini-disk 43
Chapter 10. How to SAVE a program that you write 51
Chapter 11. How to put DOS on a mini-disk. 55
Chapter 12 How to put BASIC on a mini-disk. 59
Chapter 13 How to DELETE a program or a file. 63
Chapter 14 How to DELETE a file from a mini-disk. 67
Chapter 15 How to COPY a BASIC program from one mini-disk to another 71
Chapter 16 The LOOP 75
Chapter 17 SUBROUTINES. 79
Chapter 18 The ON statement 83
Chapter 19 The TAB function 89
Chapter 20 The RANDOM function - FUN and GAMES. 93
Chapter 21 The SIGN function. 99
Chapter 22 The CHAIN command. 107
Chapter 23 The DATA statement 109
Chapter 24 The NUMERICAL FORMAT 113
Chapter 25 The EXIT statement 119
Chapter 26 Numerical LISTS and ARRAYS 127
Chapter 27 The SUBSTRING. 133
Chapter 28 SEQUENTIAL files 137
Chapter 29 How the computer READs a TYpe 3 DATA file. . .147
Chapter 30 How to ACCESS and use SEQUENTIAL files 155
Chapter 31 A closer look at the TYPe function 161
Chapter 32 How to COPY a DATA file from one mini-disk to another 167
Chapter 33 How to RANDOM ACCESS data files. 171
Chapter 34 DECIMAL and HEXADECIMAL. 193
Chapter 35 SECRETS. 203
Chapter 36 The NEW BASIC 209

Assorted Programs 219

acknowledgment

I wish to acknowledge the following individuals, whose assistance and encouragement provided much of the basis for this book.

William A. Rogers
Bob Johnston
Hillis Rogers
Barbara Rogers
Roxanne Rogers
Matt Barkley
F. Don Cooper

And a host of others......

introduction

One of the most difficult tasks for me, was setting up the order of the chapters in this book. Chapter one was no problem, as it tells you how to turn the machine on. From there on, one needs to be aware of multiple things, all at the same time.

In an effort to cope with this situation, I suggest that one does not become obsessed with wanting to understand every detail of every action, but to concentrate on the subject at hand. Before you reach the end of the book, I promise you that those unanswered questions will be answered, most of the time.

Each chapter has a major topic, indicated by the title of that chapter. In addition, certain other topics are also discussed as they relate to the major topic.

Some topics are repeated and discussed in several chapters, if it is felt that further discussion will reinforce the major topic, or if under different circumstances a rediscussion will expand the understanding of the secondary topics.

Other topics are discussed several times, when I felt that they should be explained separately and then together, in order to better illustrate their differences.

The last chapter, regarding the NEW BASIC (Release 4; June, 1978) from North Star, does not attempt to fully cover the subject, but is meant only as an introduction to it.

All the programs at the end of the book are programs that I wrote along the way, as I was learning. They are the result of much "trial and error" programming. It was through these programs that I actually began to understand BASIC.

I recommend putting these programs in your computer and RUNning them. Study them, to better understand how they work. There are many programming techniques that I have since learned, which would make these programs more efficient and more professional. I have purposely not made the changes, as I felt it would be an advantage to the reader to see the originals, in their elemental state. Even as they are, they work pretty good. At the time I thought they were great.

A method that I have tried, that makes this book a more useful tool, is to copy every program in the book and SAVE it on a mini-disk. The name for each program should be the page number, when two or more programs appear on the same page, make one the page number plus A, and the other the page number plus B. Also create any DATA FILE that is used. All of this will fit on a single mini-disk.

I have tried to make simplicity the "guide word" for this book. The working title was, "The Over-Simplified Method of Learning BASIC." Even though the title changed, my approach did not.

1

HOW DO WE GET STARTED

In most cases whenever one gets a new piece of equipment the first thing that the manual tells you, is to completely read and understand the instructions before trying to operate the equipment. If you were to do that with a mini-computer, you would never turn it on. "When all else fails read the instructions," is not the by-word when it comes to computers. The instructions are written for people who already know how to operate the equipment.

It is my opinion that "computer people" are not secure enough among their peers to write something at the level that the man on the street can understand. They seem to be constantly dropping words in their explanation like ROM and RAM, or CPU, or 8080A SUPPORT, or PROM, or the hundreads of other "in" words that they know full well that we don't know the meaning of.

I may be overly zealous in my evaluation of "computer people." It could be that terms and expressions they use in all their explanations, are part of their everyday vocabulary, and they may not even be conscious of it. If this sounds like "sour grapes" it's not. These people have so much to say, so much that they could impart upon the masses - they have seen over the mountain, but can't seem to tell us about it.

Every time I sit in front of my "television typewriter," I am frustrated by the fact that I don't know how to use this marvelous machine to the maximum of its ability. I am only limited to the maximum of my ability. There is a vast mountain between the two --- I have yet to see over the mountain. On occasion I have been permitted a peek and the prospects of more are thrilling.

If you think that I am in awe of these "mini-computers" you are right. One would normally become less entranced with something the more they knew about it ---- not so with computers. I can't help reminding myself that what I have is the "baby," the "mini" of the line. I can't conceive all that the "big daddy" can do.

Before it can do anything you've got to learn how to get it started.

YOU ARE THE MASTER OF THE PLUG.

Here is how:

This is a step by step procedure for the set up that I have. Someone who does not have the exact same equipment may consider the initial section to be useless to them, but this is not so. All computers, like all cars, have the same knobs. They may be located in different places on the dashboard, but they are there.

Assuming that everything is plugged in (the terminal, the disk-reader, and the monitor) and that all three are properly connected to the computer, do the following:

1. Press the square red button (white on some models) on the back right of the terminal, facing the keyboard. The button will lock in the "on" position. You should hear what sounds like a slight thump and then a hummmmm. This is the cooling fan.

2. Turn on the monitor (t.v. set) by pushing the rocker switch to the "on" position.

3. Turn on the disk-reader by flipping the toggle switch to the "up" position. The small red light on the front panel will not come on. It lights only when the disk-reader is actually running.

Everything is now "on." You should have a " > " sign in the upper left corner of the monitor. The rest of the screen should be blank. The " > " sign can be considered the "start up" sign. When ever it appears in that position, it indicates that the computer is neither in the Disk Operating System (DOS) mode nor the BASIC mode.

4. The computer is now ready to EXecute the command to load the Disk Operating System -- DOS. Press the UPPER CASE key located on the lower left of the keyboard. The red light on the key should be "on."

5. Place the "mini-disk" which contains DOS in the disk-reader, face up with the end which has the "oval opening" going in first. Then close the "flip latch" on the front or the disk-reader, to lock the mini-disk into place.

6. Type in the command: EX E900

Note: There is one space between the "EX" and the "E900", this exact format must be used.

This "command" will cause the disk-reader to switch on when the RETURN KEY is pressed, as evidenced by the red light on the disk-reader coming on and the humming sound which the unit will make. It will run for about 3 to 5 seconds and then cut itself off. The character " * " will appear on the upper left corner of the monitor. None of this will happen until you "enter" the EX E900 command into the computer. To "enter" anything that you type on the keyboard, you must follow it up by pressing the RETURN key.

So:

7. Press the RETURN key.

If the above described activity does not take place, you probably forgot to turn on the disk-reader. DO NOT turn it on now. The disk-reader or the computer should never be turned on or off while there is a "mini-disk" in the unit. It can cause unpredictable results. It may cause the computer to LOAD the program in the wrong place; it may WRITE strange things on the "mini-disk" in the middle of an important program; it may erase the entire "mini-disk," or if you are lucky, it may not do any of the above.

If you find yourself in this predicament; first, take the "mini-disk" out of the disk-reader. Second, *** TURN ON THE DISK-READER *** Then, press the UPPER CASE key and the REPEAT key at the same time. This is called a RESTART. The " > " character should appear on the upper left of the monitor. If it does, just start all over again. If it doesn't, take the mini-disk out of the disk reader, and turn the computer off for a second. This will wipe the slate clean. Then turn it back on, and start all over again.

8. When the character " * " appears on the monitor type in the command GO BASIC. Or to put it another way:

   On * type in: GO BASIC

9. Press the RETURN key.

Again the disk-reader will come on and run for a few seconds and shut itself off. The word READY will now appear in the upper left corner of the monitor.

You have now LOADed in DOS (disk operating system) which is a machine language program that tells the computer how to operate the disk-reader to read and write (LOAD and SAVE).

You have also LOADed into the computer the BASIC language program which allows you to WRITE and RUN programs and everything else that BASIC does.

The computer is READY. The next "move" is up to you.

2

HOW TO WRITE A PROGRAM

There are no simple answers to very complex questions, but there are very complex answers to very simple questions. For someone to ask, "How do you write a computer program?", and want only a simple answer, means that they really didn't want to know in the first place.

Learning to write computer programs is like learning anything else. You start at the most elementary level and slowly progress to the point that you realize that you will never learn as much as there is left to learn. But you keep chasing that "carrot," just like the donkey with the carrot hanging from a string a few inches from his nose, the string being attached to a pole tied to his back.

It's worth the effort, so let's get started.

To write a computer program in BASIC all you have to do is type a number, which becomes known as the LINE NUMBER, and type a STATEMENT beside it. The number can be any number from 0 to 65,535. The second LINE you write will either be executed before or after the first LINE depending on whether or not the LINE NUMBER you gave it was numerically higher or lower. The higher LINE NUMBER will be executed after the first LINE and the lower LINE NUMBER will be executed before the first LINE. If you give the second LINE the same LINE NUMBER as the first LINE, the second LINE will replace the first LINE.

A typical BASIC program is written in a "10X series" of LINE NUMBERS starting with LINE 10, then LINE 20, LINE 30, LINE 40....... This allows for ease of numbering and also permits inclusion of additional program LINEs between previously written LINEs. The importance of this will become obvious as you start to write programs.

Example:

```
10 LET A = 10
20 PRINT A + 5
```

This is a complete BASIC program that can be executed (RUN) by the computer. The meaning of the words in the program have the same meaning as they do in everyday conversation. We are going to LET A equal 10, and then we ask the computer to evaluate the numerical expression A + 5, and substitute the numerical value for A, and then PRINT the answer.

The way that we make the computer execute the program is by :

1. Type in: RUN
2. Press the RETURN key.

If we RUN we get:

```
15
READY
```

Obviously the answer is correct. The READY at the end of our answer indicates that the computer is READY for more. We can take the program that we have above and make a change in LINE 10 to expand the range of its usefulness. Let's say that instead of always making A equal to 10 we want to "put in" a value of A. So we change our program to:

```
10 INPUT A
20 PRINT A + 5
```

Where "put in" and INPUT have essentially the same meaning, except one is "computer talk."

If we RUN we get:

```
?
```

The question mark which popped up on the screen means that you know something that the computer doesn't know. It wants you to "put in" the value of A. It can not RUN the rest of the program until you do. All you need do is type in any number. Then after you type in the number, enter it into the computer by pressing the RETURN key.

On ? we type in: 240

We get:

```
245
READY
```

All the words that we have used so far in writing our program are called RESERVED WORDS. They mean exactly what they say. Since the English language has many words with the same meaning, for the sake of the computer it has been agreed to use only one of the words with that meaning. It is then called a RESERVED WORD.

The RESERVED WORD that was chosen to mean "reading matter produced from type passed through a press or an electronics device" is PRINT. No other word with that same meaning will be used.

There are about 30 RESERVED WORDS in the vocabulary of a computer in the BASIC mode. Everything that the computer does when it is RUNning a BASIC program is guided by this very limited vocabulary. Once you have "limited" your vocabulary to these 30 words when you are "talking" to the computer, you will find that it can understand you. The only problem then is to be sure that what you tell it to do, is what you want it to do. The computer is very obedient, it does "exactly" as it's told.

Without going into the meaning of this BASIC vocabulary, I shall list most of it for you to see:

1. LET
2. PRINT
3. INPUT
4. IF ... THEN
5. FOR
6. GOTO
7. ON
8. NEXT
9. STOP
10. END
11. READ
12. DATA
13. EXIT
14. RESTORE
15. GOSUB
16. RETURN
17. FILL
18. STEP
19. OUT
20. BYE
21. RUN
22. LIST
23. LINE
24. LOAD
25. SAVE
26. EDIT
27. NULL

There are several others, but these are the most often used. The one thing that they all have in common is they mean exactly what they mean in "every day talk." If you were going to explain to sombody what a program is doing, step by step, your choice of words would probably be exactly the same.

Now back to our program:

```
10 INPUT A
20 PRINT A + 5
```

If we want to test our program with several values of A, but don't want to keep typing in RUN we can add another LINE and have the computer go to the start after it finishes going through the program. So we add:

```
30 GOTO 10
```

Which tells the computer when it gets to this LINE to go to LINE 10.

If we RUN we get:

```
?10
 15
?20
 25
?2873
 2878
```

This could go on for ever. We are into a program LOOP which goes from LINE 10 to LINE 20 to LINE 30 to LINE 10 to LINE 20 to LINE 30 to LINE 10 ...... and it will keep going and going and going. The only way to get out is to "abort" the program, short of turning the computer off---which always works. To do this:

Press the CTRL key and the C key at the same time.

This is called a CONTROL-C. It STOPs a program in progress. It will interupt the program when the program finishes executing the LINE that it is on. You may have to do several CONTROL-C procedures, one after another, until the computer gets the signal when it is between LINEs. When it does, it will STOP and tell you where it STOPped. It will PRINT something like the following:

```
STOP IN LINE 30
```

Every program should not be a crisis. We should be able to get out of a program when we want to, and we can. Here's how:

Let's add a LINE or two, which will allow us to exit the program any time we want. This is what we add:

```
15 IF A = 0  THEN  40
```

Which says exactly what it says:

If the value of A equals zero then go to LINE 40

And then we make LINE 40 say END this program. We do it like this:

```
40 END
```

Now let's look at our program:

```
10 INPUT A
15 IF A = 0 THEN 40
20 PRINT A + 5
30 GOTO 10
40 END
```

If we RUN we get:

```
?25
 30
?25000
 25005
?0
 READY
```

That's basically what program writing is all about. You now can write a computer program. You can add things to it to make it do more. The remainder of this book will not be devoted to show the reader how to do something she has now learned. The rest of the book will amplify and expand this talent you now possess.

3

LINE NUMBERS

LINE NUMBERS are the "road map" that your computer follows through your programs. They don't tell the computer what to do, the LINE STATEMENT does that, they just get the computer to the right place in your program.

All BASIC program STATEMENTS start with a LINE NUMBER. The LINE NUMBER tells the computer in what order to execute the program. The computer will start with the lowest numbered LINE and proceed in increasing numerical order to the highest numbered LINE. The sequence of numbers is not important, as long as they are in increasing numerical order. LINE NUMBERS can be any whole number between 1 and 65535.

Most programmers number their STATEMENTS starting with LINE 10 and number subsequent LINEs in a 10X series.

Thus:

10,20,30,40,............65510,65520,65530

You could just as easily number your LINEs -- 1,2,3,4,...... However, this is not advisable since it does not allow "space" to insert additional STATEMENTs between existing LINES. The importance of this will become more obvious when you write your first program.

I apologize that there is not more to say about LINE NUMBERS, but there is not..........

4

HOW TO DISPLAY A PROGRAM

How to see what you got is important, so that you know what you have. This is called LISTing a program that is in the computer. The program either got there by writing it while the computer was in BASIC mode or it was LOADed in from some "device." The "device" in my case would have been a "mini-disk."

For the purpose of the following discussion let's assume that whatever we ask the computer for is there, unless said otherwise. Our first question to be answered is:

How do you display a program on the monitor (t.v.) once it has been written or LOADed?

1. On PROMPT or READY type in: LIST
2. Press the RETURN key.

The entire program will be displayed on the monitor. If the program is longer than the number of lines the monitor will display at one time (16 lines) the program will scroll by starting from the lowest numbered LINE to the end of the program.

If you want to display only a portion of a program and you know the LINE NUMBERs involved you can ask for only that portion. For example:

You have a program with LINE NUMBERs from 10 to 1200 in a 10X series , i.e. 10,20,30,---1190,1200. You want to look at that portion of the program from LINE 820 to LINE 930.

1. On PROMPT or READY type in: LIST 820,930
2. Press the RETURN key.

The computer would then display all the LINEs between and including LINE 820 and LINE 930, a total of 12 LINEs.

If you then wanted to scan the last half of that same program, i.e., from LINE 600 to the end, you would:

1. On PROMPT or READY type in: LIST 600
2. Press the RETURN key.

The computer would then scroll by on the monitor, starting at LINE 600, the rest of the program to LINE 1200; the end.

If you wanted to look at only LINE 600, you would:

1. On PROMPT or READY type in: LIST 600,
2. Press the RETURN key.

The computer would display only LINE 600 on the monitor.

NOTE: The only change between the above example procedure and this one it the addition of the "comma."

NOTE, NOTE: If you are sitting in front of your computer, and you just tried all of the above examples,and everything having to do with LISTing worked, but seemed to come out backwards, give yourself one gold star for noticing. This tells you that you have the "new" BASIC. From this point until you reach chapter 36, the above LISTing example will be one of two, of the only indicators of this fact. With the exception of the COmpact command, it will also be the only difference. Everything else in this book, as written, applies to your version of BASIC.

The above procedures are the most salient features of LISTing a BASIC program that is in the computer. However, there is another type of LISTing that is also important. This is the LIsting of the "files" that are on the "mini-disk." This procedure is not done in the BASIC mode but is done in the "disk operating system" or DOS mode.

If you had a "mini-disk" and you wanted to know the names of the files on that "mini-disk" you would do the following

1. Put the computer in the DOS mode.

   a. From start up:

      Put the mini-disk with DOS in the disk-reader.

      On > type in: EX E900

      Press the RETURN key.

   b. From BASIC mode:

      On READY type in: BYE

      Press the RETURN key.

You know when the computer is in the "disk operating system," i.e., DOS, because of the " * " sign. If it is the last item to appear on the monitor at the far left --- you are in the "disk operating system" -- DOS. If the word READY is the last item to appear, you are in the BASIC mode. If just the prompt (cursor) is the last item to appear you are generally in the BASIC mode. If the " > " sign appears in the upper left corner and the rest of the screen is blank you are not in either mode, you are at start-up.

2. Put the "mini-disk" that you want LISTed in the disk-reader.

3. On * type in: LI

4. Press the RETURN key.

You will see displayed on the monitor a complete LIsting of all the files on that "mini-disk."

Now that's the last word on LISTing and LIsting, but not the last word on "seeing what you got." All those neat little columns of numbers that are located next to the names of the files mean something. But, we won't get into that till later.

5

VARIABLES

If you were to ask "real computer people" what they thought of BASIC, they would look down their noses at you and tell you that they don't see how anybody can make sense out of all those "dollar signs" and "C5's" and all that stuff. "It's nothing but a jumble of ABC's," they'd tell you.

Don't be upset or taken in by their condescending attitude. Put their remarks in perspective. Don't forget that many of them consider the mini-computer a "toy computer." Without their taking the time, or making an effort to see what the "state of the art" is with mini-computers, they just assume that all they are are glorified pocket calculators.

Having had the opportunity to expose several "real computer people" to the job that my mini-computer is doing for my business, in accounting, quality control, shipping, product records, and also as a "word processor," they are usually very surprised at what they see, and leave with a whole new attitude about the mini-computer.

But, irrespective of their initial attitude, there is some merit in what they say about all those ABC's. Most BASIC programs RUN on ABC's, and if you don't keep up with them, your program can turn into a "wild jumble" of "dollar signs" and "C5's."

There are 286 possible variable designations for any single numerical value. You can assign any letter of the alphabet to represent a numeric variable, the choice is yours - from A to Z. In addition to any letter of the alphabet, you can combine your chosen letter with any number from zero to nine. Some of the possible choices are:

K , N6 , L0 , P8 , D , G4 , Z9 , W5 , T2 , O7

This system gives you 286 variable "names" to represent numeric variables --- there are no rules, pick any one you want, anytime you need one.

The very same set of rules apply to the system of assigning "names" to STRING variables (WORDS), with one addition, you must add a "dollar sign" ($) as the last character of the variable "name." Some typical designations for STRING variables (word variables) are:

A$, N6$, L0$, Y2$, J7$, C$, P9$, S8$

Counting the 286 numeric variable "names," plus the 286 STRING variable "names," you have random access to a total of 572 "names" to choose from --- as the man said, "that's a lot of ABC's."

Although you need not be frugal in your use of variable names, nor do you need to be very selective, you should choose them in some orderly fashion so as to prevent using the same variable name in a given program to represent two different variables. The method you choose is entirely up to you, there is no standard method of assigning variable names.

If one is writing a very long program, and it is taking days or weeks to write, or to complicate it further, two or more people are working on it, I would recommend using a variable tally sheet. There are actually two tally sheets, one for numeric variables and one for STRING (word) variables.

These tally sheets list all the possible variables, and when you use one, all you do is cross it off the sheet. This will preclude the chance of using the same variable more than once. In a situation as I have outlined above, I would not rely on your memory; it's not as good as the computer's when it comes to such details. You may forget that you assigned the same variable name to two different variables, but the computer won't.

Here are the programs which will generate your tally sheets:

```
10  DIM A$(26),B$(26)
20  !"     N U M E R I C  V A R I A B L E  L I S T  "
30  !
40  FOR  I = 1 TO 26
50  FOR  J = 1 TO 11
60  LET A$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
70  LET B$ = " 0123456789"
80  ! A$(I,I),B$(J,J),"   ",
90  NEXT J
100 !
110 NEXT I
```

And:

```
10  DIM A$(26),B$(26)
20  !"     S T R I N G  V A R I A B L E  L I S T "
30  !
40  FOR I = 1 TO 26
50  FOR J = 1 TO 11
60  LET A$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
70  LET B$ = " 0123456789"
75  LET T$ = "$"
80  ! A$(I,I),B$(J,J),T$,"  ",
90  NEXT J
100 !
110 NEXT I
```

Contrary to some published advertisements, that is not all that you need to know about variables, but that's a start....

6

THE CAST OF CHARACTERS

These small, seemingly insignificant, things are the mortar that hold the whole BASIC LANGUAGE together. The comma, the quotation mark, the exclamation mark, the back-slash, the per-cent mark, the dollar sign, the number sign, the at-sign, and the space. All of these characters play a major role in the BASIC LANGUAGE. The presence or absence of a single one of these characters can make the difference between a well RUN program and absolute garbage.

We shall look at each one of these and discuss their most salient functions. I will not at this time attempt to cover all the characters, some for which I can not even find a name, or their function. I'm not sure that any single person is familar with all of them --- I'm not. There are a few that you must become well acquainted with in order to work with BASIC. Here are the more ubiquitous.

THE COMMA

. . . .

The COMMA is the "work horse" of the BASIC LANGUAGE. It is used to separate variables, limit procedures, format OUTPUT, separate system instructions, segment statements, and on, and on, and on....

Here are some examples of the more common uses for the comma:

```
  10  READ #0, A,B,C$,D,E7$
  10  INPUT A,B$,X
  10  ! "THERE ARE ",K," INVOICES OUT"
  10  PRINT A,
  10 !"THERE IS STILL ",%$C8F2,G," IN THE ACCOUNT "
  10 INPUT " DISCOUNT CODE 1,2,or 3 : " R7
  10 ON J GOTO 125,160,240
  10 OPEN #0, "AUTOPART"
  10 DATA 1,22,3,44,5,66,7,88
  10 WRITE #0, A,B$,C,D$,F7$
  10 LET N = R(B,0)
  LIST 250,300
  RUN 140,200
  LOAD AUTOPART,2
  SF AUTOPART,2
  CR DOS, 1 10
```

These examples show most of the uses of the COMMA that you will encounter.

The ability to format OUTPUT is a use that you must become familar with at the onset. If a PRINT statement is not terminated by a COMMA, the OUTPUT from that statement is printed on a single line by itself.

Thus:

```
10 FOR S = 1 TO 4
20 PRINT S
30 NEXT S
```

If we RUN we get:

```
1
2
3
4
```

If we change LINE 20 to:

```
20 PRINT S,
```

If we RUN we get:

```
1 2 3 4
```

For the program:

```
10 FOR X = 1 TO 4
20 PRINT X
30 C = X+10
40 PRINT C
50 NEXT X
```

If we RUN we get:

```
1
11
2
12
3
13
4
14
```

If we change LINE 20 to:

```
20 PRINT X,
```

If we RUN we get:

```
1 11
2 12
3 13
4 14
```

If we then change LINE 30 to:

```
30 PRINT C,
```

If we RUN we get:

```
1 11 2 12 3 13 4 14
```

What is happening is that the COMMA at the end of a PRINT statement tells the computer to "add" the OUTPUT from the next PRINT statement to the same line. If all the PRINT statements in a given program had COMMAs at the end of them, then all the OUTPUT would be put on the same line until it overflowed to the next line. If none of the PRINT statements had a COMMA at the end, then all OUTPUT from each PRINT statement would be on different lines.

If you have a program in the computer's memory that is 150 LINEs long, and you want to look at LINEs 20 to 110, if you typed in LIST the entire program would scroll by unless you were fast enough to execute a CONTROL C in time to STOP it at the part that you wanted. That's an iffy situation: sometimes you can; most times you can't. The best way to do it is:

On READY type in: LIST 20,110

The computer will then display LINEs 20 to 110 and STOP by itself. If you want to look at the last part of the program, from LINE 110 to the end, then:

On READY type in: LIST 110

Note, no comma.

If you just want to look at LINE 110 and nothing else, then:

On READY type in: LIST 110,

NOTE: If you have a problem with the above, see chapter 36.

Most of the other uses of the COMMA are illustrated in the above examples. Their effect is not as dramatic as the discussed examples, but just as significant. Proper placing of the COMMA in a LINE statement is essential. Be sure that you separate all variables in LINE statements by COMMAS, regardless of whether it is a DATA, a READ, a WRITE, an INPUT, a PRINT, or an ON statement. If there is more than one variable or value on a LINE, it needs a COMMA between it.

On the other hand, don't use too many COMMAS. The last item on the above mentioned LINE statements should not be followed by a COMMA, with the exception of the PRINT statement, then it's optional. If you are in doubt about whether to use the COMMA or not, check the above examples or just put it "in" and then "take it out" and see what happens.

QUOTATION MARKS

The QUOTATION MARK can not be overlooked. If ever you have anything that needs to be said, the computer will not say it unless you enclose it in QUOTATION MARKS, the computer does not want to be responsible for what you have to say. The QUOTATION MARK is especially important in formating OUTPUT, allowing the computer to identify and handle STRINGs (words), and allowing the computer to differentiate between numbers and numeric expressions.

Some examples of LINE statements with the QUOTATION MARKS in them are:

```
20 IF G$ = "YES"  THEN 80
20 OPEN #0, "FRUIT"
20 INPUT "WHAT IS THE INVOICE NUMBER : ", N
20 PRINT "THE VALUE OF X IS : ", X
20 !"PART NUMBER ",K," IS NOT IN STOCK "
20 !"THERE ARE ",J," INVOICES WITH A TOTAL OF ",R
20 ! A,"    ",B,"    ",C,"    ",D,"    ",E
20 !" KRISTI MAY SUE "
20 DATA 5,"HY-TEMP SEALED HEADLIGHT",6,"SPARK PLUG"
20 !"A=",A," B=",B," C=",C," D=",D
20 !" GROSS INCOME = ",%$C10F2,G
20 !"******************************************"
20 !"I---------I----------I---------I--------I-------I"
20 !"3+5=",N
```

All of these are practical uses of the QUOTATION MARKS . All of the above examples came from actual programs, most of which are in this book. Note the flexibility that is allowed when using the QUOTATION MARKS. You may PRINT an entire line merely by enclosing its contents with QUOTATION MARKS, or you may segment a LINE as many times as space allows, for the inclusion of a variable.

Anything that you enclose by QUOTATION MARKS in a PRINT statement becomes a STRING (a word) to the computer. If it is a numerical expression it is not evaluated by the computer, it is just PRINTed, exactly as you enclosed it.

For example:

If on READY you typed in: PRINT 3 + 5

You would get: 8

but, if on READY you typed in: PRINT "3 + 5"

You would get: 3 + 5

The difference between the two examples is that without the QUOTATION MARKS the 3 + 5 is a numeric expression and the computer is programmed to solve the expression and PRINT out the answer. The " 3 + 5 " enclosed in QUOTATION MARKS is a "word", and the computer is programmed to PRINT out anything enclosed in QUOTATION MARKS exactly as it appears.

Consider this program:

```
10 FOR Y = 1 TO 4
20 PRINT Y,
30 NEXT Y
```

If we RUN we get: 1 2 3 4

If we change LINE 20 to:

```
20 PRINT "     ",Y,
```

If we RUN we get: 1 2 3 4

Note the spaces between each number. The computer PRINTs what ever is between the QUOTATION MARKS: in this case it was blank spaces.

If we were to change LINE 20 to:

```
20 PRINT "THE VALUE OF Y = ", Y
```

If we RUN we get:

```
THE VALUE OF Y = 1
THE VALUE OF Y = 2
THE VALUE OF Y = 3
THE VALUE OF Y = 4
```

Other than for a few examples at the start of this section, I have not gone into one of the most important functions of the QUOTATION MARK, its role in handling STRING VARIABLES or in a loose definition, WORDS. As far as the computer is concerned, if it is not a numeric value or a numeric expression to be solved --- it's a WORD (STRING).

That being the case, the computer wants you to keep your "words" and your numbers separated. A numeric variable is represented by a single letter of the alphabet, which may or may not be coupled with a number from zero to nine -- G4. A STRING variable is represented by a single letter of the alphabet, which may or may not be coupled with a number from zero to nine, plus a dollar sign ($) -- G4$. In this regard, there is one hard and fast rule which you must remember, that is:

ALL STRINGS (WORDS) WHICH APPEAR IN THE BODY OF A BASIC PROGRAM MUST BE ENCLOSED IN QUOTATION MARKS.

THE BLANK SPACE

A quick general statement that comes to mind about BLANK SPACES is that the computer completely ignores them. But, like most "quick general statements," that's wrong. If I were to modify that to, the computer usually ignores them, that would be closer to the fact.

The computer does not allow BLANK SPACES in the following:

1. In reserved words. These are words which have a special meaning to the computer. They are either "command words" or "statement words." When they appear in the body of a BASIC program they must remain intact if they are to mean anything special to the computer. Some examples are:

STATEMENTS:

GOTO	ON ....GOTO	IF.....THEN
GOSUB	RETURN	RESTORE
FOR	NEXT	END
STOP	LET	READ
WRITE	OPEN	CLOSE
INPUT	FOR....TO	PRINT

COMMANDS:

LOAD	BYE	EDIT
LIST	RUN	PRINT

2. You can not have BLANK SPACES in the "name" of any FUNCTION, such as:

    ABS(G)   RND(J)   INT(T)

3. You can not have BLANK SPACES in any LINE NUMBER, such that:

```
                        50  GOTO  110
is not the same as
                        5 0  GOTO 110
```

The first example being read by the computer as LINE 50 and the second as LINE 5.

```
                        50  GOTO  110
is not the same as
                        50  GOTO  1 1 0
```

The first example being read by the computer to GOTO LINE 110 and the second to LINE 1.

4. No BLANK SPACES can be used in a "defined" STRING (word), such as a FILE NAME or a STRING match, without it changing the STRING.

```
HAPPY   is not the same as  H A P P Y
REDWINE is not the same as  RED WINE
```

All of the above forms are acceptable. You must, however, be cognizant that they are not the same.

Now, back to my original statement, "The computer ignores all BLANK SPACES, excluding the above mentioned exceptions." That still leaves an opportunity for lots of wide open spaces or if you prefer you can do away with BLANK SPACES altogether.

Consider the following two programs:

```
10   FOR   X   =   1   TO   3
20       Y           =         X      +     1
30              PRINT                  Y
40                          NEXT        X
```

and:

```
10FORX=1TO3
20Y=X+1
30PRINTY
40NEXTX
```

This is no doubt a silly example, but it does illustrate the latitude with which one has to spread out a program or condense it to his own liking. Both programs are properly typed and the computer would RUN either with no problem.

THE BACK SLASH

Since we are talking about condensing things, let's next discuss the "Borden's" of condensed BASIC computer programs -- the BACK SLASH. The BACK SLASH allows you to put as many STATEMENTS on a single LINE as you can possibly squeeze in. The computer will execute each STATEMENT in order from left to right, just as if each STATEMENT had its very own LINE NUMBER, each a little bit higher than its neighbor to the left.

Consider the program in the section on BLANK SPACES where we took out all the spaces, that same program could have been condensed even further by the BACK SLASH. It would look like this:

```
10FORX*1TO3\Y=X+1\!Y\NEXTX
```

This is the very same program. It will RUN just like the above mentioned program, and yield the same OUTPUT. To the computer there is absolutely no difference.

Why then, you say, do we go to all the trouble to spread things out, use all those LINE numbers, take up all that space, all that memory, if the above example is exactly the same to the computer? The answer is that we don't have to, but if you ever had a problem with a long program written as above, you would go batty trying to figure out what it says. Unless you were very lucky, or unless you knew exactly where your error was, you would have to go back and spread it out in order to see what you had.

Many professional programmers, once they have completely "debugged" a program, will then go back and "compress" their program to discourage other people from copying it, or changing it, or figuring it out.

THE AT-SIGN

The AT-SIGN is of little or no value to the person who does not make mistakes, not even typing errors. For the rest of us however, it provides a quick, convenient method to "start all over." Whenever you press the AT-SIGN key -- " @ ", it will cause the computer to void whatever was previously typed on the current LINE, and jump back to start that LINE all over again.

For example:

```
10 ! " WHAT'S UP DOT @
```

This whole LINE will be voided and you can start all over again.

THE EXCLAMATION POINT

I have saved the "character" that I like best, for last. The EXCLAMATION POINT has saved more time, more bytes, and has made palatable a portion of what I consider to be the drudgery of computer programming --- typing.

I have a problem in that I can't type as fast as I can think. If I try, I find that what I typed is not what I thought. Any BASIC computer program "worth its salt" is peppered with PRINT statements.

All that the EXCLAMATION POINT does, is allow you not to have to type the word PRINT. You can type an EXCLAMATION POINT in its place. It will replace the word PRINT either as a statement in a LINE or as a command to the computer, any place you would use the word PRINT, you can replace it simply by typing an EXCLAMATION POINT.

```
10 PRINT A,B,C   is the same as    10 ! A,B,C
 PRINT FREE(0)   is the same as     ! FREE(0)
```

That's all that it does, but I'm glad that it does that....

7

THE RENUMBER FUNCTION

On the surface this hardly seems a worthy enough subject to give special attention, and it may not be. But, I have found this little RENumbering "function" is able to bring a little class and order to a program, and it only takes a second.

All the RENumbering function does is renumber a program. It makes the first LINE of a program LINE 10 and each subsequent LINE number 10 more than the last LINE number. This is what I refer to as a "10X series," i.e, 10,20,30,.... .... 65520,65530. That's as high as it goes.

That is not all that the RENumber function renumbers. If there is a LINE number in the content of a LINE it will also change that number to match the new number assigned to that LINE. I find this pretty amazing when I consider that in almost any program you will find at least 10 or more "GOTO's," or "IF/THEN's," or even more complicated, an occasional ON statement.

The time to use the RENumber function is after you've done all the adding of LINEs, EDITing, DELETing, and just before you SAVE the program. RENumbering makes future editing and tracking of the program a lot easier to work with when looking for errors, or making additions. It's worth the effort and should be done.

Let's look at a few examples:

```
1    REM  THIS MAKES A MATRIX  5 X 6
6    DIM M(5,6)
10   FOR J = 0 TO 5
15   M(J,0)= J
23   FOR K = 0 TO 6
31   M(0,K) = K
40   PRINT M(J,K),
51   NEXT K
52   PRINT
63   NEXT J
70   END
```

This program was written by one of the most used methods in computer programming today. The method was the "trial and error" method. Most programs written by this method, which have not been

RENumbered, usually have a strange assortment of LINE numbers. This is the result of constantly changing, editing, adding, and deleting of LINEs until you get a program that works.

So that nobody will know how much trouble you had writing a program, you should RENumber it. Do this:

1. On PROMPT or READY type in: REN
2. Press the RETURN key.

Now to see the results do a LIST.

```
 10  REM THIS MAKES A MATRIX  5 X 6
 20  DIM M(5,6)
 30  FOR J= 0 TO 5
 40  M(J,0) = J
 50  FOR K = 0 TO 6
 60  M(0,K) = K
 70  PRINT M(J,K),
 80  NEXT K
 90  PRINT
100  NEXT J
110  END
```

Note that all the LINE numbers are now in a "10X series" starting with LINE 10 and that the LINE number in the body of the program (LINE 100) has been appropriatly changed.

The same RENumber function can also be used to expand LINE space in a program. Aassume that the following extreme example is the result of the "trial and error" method:

```
 .
 .
 .
 .
520 IF X = 25 THEN 523
521 IF X = 50-C THEN 524
522 IF X > 25  AND  X < 50 THEN 525
523 GOSUB 1248
524 LET D = 22
525 GOTO 60
.
.
.
```

Let's suppose that after checking the program we find that if the program returns to LINE 524 after it finishes its subroutine from LINE 523 it will not RUN. So all we have to do is add a GOTO statement so that it dosen't come back to LINE 524. But wait, we've boxed ourselves in. There are no numbers left that we can use to add a LINE at the required place. All we have to do is:

1. On READY type in: REN
2. Press the RETURN key.
3. On READY type in: LIST

We will have to scroll through the program and look for the desired section that we are interested in since we do not know what the new LINE numbers are to LIST to. Let's say that we find them and the new numbers we want are 1150 to 1200. So we do the following:

1. On READY type in: LIST 1150,1200

and we get:

```
1150 IF X = 25 THEN 1180
1160 IF X = 50-C THEN 1190
1170 IF X >25 AND X < 50 THEN 1200
1180 GOSUB 1550
1190 LET D = 22
1200 GOTO 80
```

Now we can add one LINE or up to nine additional LINEs after our GOSUB statement if we want to. Problem solved.... Text on RENumber function ended.

8

HOW TO CORRECT OR EDIT A PROGRAM

I'm sure that the second thing that was done after the computer was worked out was to develop a system to correct programmer mistakes. It must have been paramount in the mind of the people who developed the BASIC language, because there are so many different ways to correct the same mistake. Sometime you spend more time trying to decide which method of editing to use than if you just did the whole thing over. I shall discuss some of the most used methods.

If we have the following LINE in a program:

```
130 IF A = B  THEN 60
```

You want to change it to:

```
130 IF A = B  THEN 110
```

You do the following:

1. On PROMPT or READY type in: EDIT 130

The EDIT command is a built in "function" which essentially treats the PROGRAM LINE that matches the LINE NUMBER after EDIT, as if it were the last thing typed into the computer.

2. On PROMPT press: CTRL key & G key

When you press the CTRL key and the G key together it is called a CONTROL-G. The CONTROL-G is a built-in edit "function" which automatically rePRINTS the last item that was typed into the computer.

Therefore you should get:

```
130 IF A = B  THEN 60
```

Now do a "shift-delete" by pressing the DEL key while holding down the SHIFT key. You will notice that each time you do a

"shift-delete" one character at the end of LINE 130 is erased. Since to make the desired change we need to erase the "60" we do the "shift-delete" twice. Now that we have erased the "60", the prompt is in the proper position to type in the number "110". After you have typed in the "110" press the RETURN key. You have made the desired change. To verify that the desired change was made, do a CONTROL-G, you should get:

130 IF A = B THEN 110

If there are no further corrections, press RETURN key.

Now let's make a change at the other end of the line, we want to change the LINE NUMBER 130 to LINE NUMBER 170.

1. On PROMPT or READY type in: EDIT 130
2. Press the RETURN key.
3. On PROMPT type in: 170
4. Do a CONTROL-G.

You have now made the desired change. If you want to confirm this do another CONTROL-G. You should get:

170 IF A = B THEN 110

And you do! However, you have left something behind. If you were to LIST your program you would find that you have LINE 170, as you just confirmed, and you would also see that LINE 130 didn't erase when you "replaced" it. When you made all the other changes, one change automatically erased the thing it was changing. LINE NUMBERS are something special. If you want to do away with a LINE NUMBER you specifically have to tell the computer that. Here's how:

1. On PROMPT or READY type in: 130
   (LINE to be erased)

2. Press the RETURN key.

The unwanted LINE 130 is now gone...

Now let's change:

170 IF A = B THEN 110

to:

170 IF A = 2*B THEN 110

Since the desired change this time is in the middle of the LINE, we can approach it from either end; it really doesn't matter. If the end results are the same, there is no "right" way. I will make the change starting from the LINE NUMBER side, you try it from the other end:

1 . On PROMPT or READY type in: EDIT 170
2. Press the RETURN key.
3. On PROMPT press: CTRL key and A key. Continue to do this until you have passed the " = " mark in your mark in your LINE.

This is called a CONTROL-A. It is another "control function" of the computer. It will automatically PRINT one character at a time - each time you press the A key while holding down the CTRL key - of the last thing typed into the computer or put in that "position" by an EDIT command.

4. Now press the CTRL key and the Y key.

This is called a CONTROL-Y. It is another "control function" of the computer. It is also called the "insert control." It is a two step "control function." The first time you do a CONTROL-Y it will print a " < " at the start of the LINE position that you executed it. Such as:

170 IF A = <

Then you type your desired change:

170 IF A = <2*

Then you do the second CONTROL-Y which will print the " > " sign at the end of you insertion. You now have:

170 IF A = <2*>

Then you do a CONTROL-G which prints the remainder of the line and you get:

170 IF A = <2*>B THEN 110

Then you do another CONTROL-G (This one is optional) and you get:

170 IF A = 2*B THEN 110

Which is exactly what we wanted.

There are at least two ways to delete characters from a program LINE. Again, both are commonly used and except under certain conditions, which one you use is up to you. Let's change LINE 170

from:

170 IF A = 2*B THEN 110

to:

170 IF A = 2 THEN 110

Method A:

1. On PROMPT or READY type in: EDIT 170
2. Do a CONTROL-A until the "2" appears.
3. Press the "space bar" for each character you want to delete, i.e,..two times.
4. Do a CONTROL-G to display the remainder of the line.
5. Press the RETURN key to effect the change.

and we get:

170 IF A = 2   THEN 110

Which is what we wanted. However, lock at all those wide open spaces. I have already told you that the computer dosen't care if they are there, so if you don't care, then everything is fine. If you do care then I recommend method B.

1. On PROMPT or READY type in: EDIT 170
2. Do a CONTROL-A until the "2" appears.
3. Press the CTRL key and the Z key for each character you want omitted (two times).

This is called a CONTROL-Z. It is another editing "function" of the computer. Each time you use it it will replace the character in that position with a "%" sign, until you are finished with all your changes and you enter the changes in the computer. When you LIST your change or do a CONTROL-G after you have EDITed, the omitted characters and the "%" sign are gone and so are the characters they replaced, and so is the "space" they occupied.

4. Do a CONTROL-G to complete the remainder of the line.

we get:

```
170 IF A = 2%%  THEN 110
```

5. Press RETURN key (this enters the change into the computer).

6. Do a CONTROL-G to see the EDITed LINE (optional).

we get:

```
170 IF A = 2  THEN 110
```

Which is exactly what we wanted.

As to those conditions when it is better to use the CONTROL-Z method of deletions as opposed to the "space bar" method of deletion, it depends on how much space you can afford to use. Even though the computer essentially ignores excess spaces in LINE items, it does not completely ignore them. It SAVEs them for you. If you put them there, it assumes that you wanted them there. Everytime it SAVEs the "blank space" it also takes up one byte of memory. There are to my knowledge no "reserved words" or "commands" which have spaces in them. Assuming that to be the case, all "spaces" could be eliminated from all programs. Thus our LINE 170 could be written:

```
170IFA=2THEN110
```

The computer will handle this LINE 170 just as easy as the one above. Besides that, you would save 7 to 9 bytes on just that one LINE. Can you imagine the almost impossible task of trying to "track" a three hundred LINE program written without any "spaces" while looking for an error?

There are other EDIT "functions" in your system. By understanding those that I have written about, you are in a better position to evaluate the use of those I have not written about. Then you can decide whether they are worth the effort.

9

HOW TO LOAD A PROGRAM FROM A MINI-DISK

There are two ways to get a program into the memory of a mini-computer. One we have already discussed; that of writing a program into the computer. The other we have already used; that of LOADing a program into the computer from a "device." That device can be a tape recorder, a paper tape, an ordinary telephone, magnetic card, or the "device" that we used to LOAD in DOS and BASIC (both operational programs), the "disc-reader."

The "disk-reader" is a wondrous machine. It can READ the magnetic "mini-disk" in a matter of seconds. In less time than that, it can "look" at what you want and "look" at what it's got, and tell you if your "wants" exceeds its "gots." If you ask for something it can not find on the "mini-disk," it will try ten times to locate the item. If it's not successful, the disc-reader will tell the computer that it's not able to locate the ask for item and the the computer will print that on the monitor. All that will take place in just a few seconds or less.

The "mini-disk" is nothing more than a large piece of tape recorder tape that is the size of a five inch saucer instead of a long skinny tape. It is essentially "played" and "recorded" in the same manner as on a regular tape recorder. The unique feature of the "disk-reader" is its ability to loacte any requested item at any site on the "five inch saucer." Each mini-disk can contain about 90,000 pieces of information (bytes). You can ask for any one piece and the disk-reader will locate it for you in a matter of seconds.

It works very much like a record player. The recording head, called the "pointer," is like the arm on the record player. The mini-disk is like the record. The disk-reader turns the mini-disk, just like the turntable turns the record, only much faster. The arm on the turntable, the "pointer" on the disk-reader, follows the grooves in the record, the tracks of the mini-disk, starting from the outside rim and working its way to the center.

Just like one of these new, modern turntables that can put the playing arm down on any pre-selected band on the record, the computer, by use of the Disk Operating System can put the "pointer" down on any position (address) on th mini-disk.

Now that you know how it operates, let's learn how to operate it.

Every program on a "mini-disk" has a name. The name that the program is given is up to the programmer within certin limatations. All activity involving this program is done by using its "given name" and nothing else.

The "given name" of a program must be no more than eight characters long. These characters can either be letters, numbers, or symbols. There are only two exception. You can not use a blank or a comma in the name.

The name should be an acronym which will tell the programmer what the program is about. The name for an "invoice preparation program" could be "INVOPREP" or the name for a program which reads the contents of a mini-disk file could be "DISKREAD". Choose the names of your programs so that they mean something to you.

Let's assume for our example that we have an "inventory accounting program" which we will name INVACCTN. We could have just as eaisly named it RABBIT or #*+4R& or 6/7/78. All of these options fulfill the namming requirement. But, let's use the name INVACCTN. We must further assume that the program named INVACCTN is recorded (SAVEd) on the mini-disk that we have in the disk-reader. On this same mini-disk is the program for our Disk Operating System (DOS) and the BASIC Language program.

After making all these assumptions and meeting all the stated requirements, we are ready to go from "start up" to putting in (LOADing) our programs to RUNning them.

Here's how:

1. To put (LOAD) the Disk Operating System (DOS) in the computer:

   On > type in: EX E900

2. To put (LOAD) the BASIC Language into the computer:

   On * type in: GO BASIC

3. To put (LOAD) the program named INVACCTN in the computer:

On READY type in: LOAD INVACCTN

The word READY again pops up on the monitor.

Each time that you "type in" something on the terminal keyboard you have to follow it up with pressing the RETURN key in order to "enter" it into the computer.

Each time you "press RETURN key" the disk-reader will come on, run two or three seconds, turn itself off, and replace the last symbol on the screen with the next required symbol on the monitor, first > then * then READY --- if all goes well.

The different symbols and the word READY all tell you that the computer accepts the LOADed program and is in the required mode for the next "command." The symbols mean:

> . . . Tells you that the computer is in the "start-up" position and there are no programs in the computer.

* . . . Tells you that the computer is in the Disk Operating System - DOS mode.

READY . . Tells you that the computer is in the BASIC mode and that you can READ, WRITE, SAVE, and LOAD BASIC programs.

If you were already in the BASIC mode and you tell the computer to do something, it will print READY after if it finishes its assignment.

Since, after we told the computer to LOAD INVACCTN we got READY. That would indicate that the computer has LOADed our program and we are READY to RUN. If you do not have "complete faith" and you want to see the program. Do this:

On READY type in: LIST

The program named INVACCTN will scroll by from the first LINE to the end. If you are now convinced that it is there and you want to RUN the program do the following:

On READY type in: RUN

When you press the RETURN key to enter the "command" into the computer, it will start executing the program.

That's the way it should always work. On occasion it dosen't. Let's consider the most frequent of these times.

If at any time you command the computer to LOAD a program and it comes back and displays on the monitor "HARD DISK ERROR" or HD 000, that means that for some reason the computer was not able to transfer the information on the mini-disk to its memory. The most frequent cause for this "ERROR" is that the mini-disk was not properly inserted in the disk-reader or that the "lock gate" on the disk-reader was not closed.

To correct this "ERROR" check the "lock gate" if it is not closed, close it and repeat the LOAD command. If it was properly closed, open it, take out the mini-disk. Make sure it was inserted "face up" with the end with the "oval cut-out" going into the slot on the disk-reader first. Even if it was properly inserted, reinsert it, close "lock gate" and re-enter the LOAD command.

If it still does not LOAD the program and HD 253, or any number following the HD appears; two things are probable, Number one, the mini-disk is damaged. If that's the case, nothing can be done and you have lost whatever program that was stored on the mini-disk at position (BLOCK) 253.

The mini-disks are not fragile, but do require some special handling. They should not be "bent, folded, or mutilated." They should not be placed in any type of magnetic field. They should be kept "high and dry" and in the envelope that came in when not in use. In general, handle them as you would an expensive phonograph record.

The second possibility is that a piece of lint or something got between the disk-reader head (pointer) and the mini-disk. Sometimes just removing the mini-disk from the disk-reader and lightly bouncing it on its edge a few times on the table or carefully turning the mini-disk in its protective holder will correct the problem. Re-insert the mini-disk and try to LOAD it again.

If it still doesn't work there are two possibilities. Number one, the mini-disk is damaged. Number two, the disk-reader is malfunctioning. To check this out try to LOAD another program from the same mini-disk. If you are sucessful then consider the "HARD DISK ERROR" is isolated to that one place (BLOCK) on that mini-disk. If not, try to LOAD a program from another mini-disk. If you are sucessful then consider that it is not your disk-reader that is at fault, but that the previous mini-disk is actually damaged. If you fail on all attempts its probably your disk-reader. You don't need more instructions --- you need a repairman.

Assuming, that none of the above happened, but you still didn't LOAD your program, the next most frequent occurrence may be the appearance of the following phrase:

ARG ERROR

This stands for "argument error" which means that what you asked the computer to do, it could not do, because what it needed to do the task was not available.

In our particular case, what it says is that there is no program on the mini-disk with the name you told it to LOAD. If we are sure that there was such a program on the mini-disk, then perhaps we made an error in our LOAD command. For instances, if we told the computer to:

LOAD INVACTN

We should expect an ARG ERROR, because the name of our program was:

INVACCTN

Look closly at the spelling. The computer never forgets what you tell it. It does exactly what you tell it to if it can. Most of us mortals have to learn how to deal with such consistency --- it's not all that easy.

Knowing that there is no room for ERROR, and our memory is not good enough to come up with the exact name of a program, and the chances of us lucking up and guessing it are slim, the "computer people" provided us with a solution. There is a method of getting the computer to LIst the names of all the programs on a mini-disk.

Here's how:

If READY is the last message on the monitor, we are still in the BASIC mode. In order to display the mini-disk "file" LIst we must get back to the Disk Operating System - DOS mode.

To do this you must:

On READY type in: BYE

When you press the RETURN key to enter this command into the computer, a * will appear as the last item on the monitor. That means we are in DOS mode.

To get our LIsting of this "file" we then:

On * type in: LI

The disk-reader should come on, and a LIst of all the programs on that mini-disk will be displayed on the monitor.

With the complete LIsting of the programs and there exact spelling in front of you, you are now able to properly LOAD the program you want. But, you cant LOAD a BASIC program in the DOS mode. Here's how to get back to BASIC:

On * type in: JP 2A04

When you press the RETURN key you will see that READY is now the last entry on the monitor. Which indicates that we have JumPed back into the BASIC mode.

Now:

On READY type in: LOAD INVACCTN

The word READY should now appear on the screen again, telling you that all is done and the computer is READY to RUN.

On occasion after the LOAD command is given the phrase:

NO PROGRAM ERROR

will appear on the monitor. What this tells you is one of two things:

Number one, there was in fact a space (file) created for a program called INVACCTN, and this LIsting is on the mini-disk, but that program was never SAVEd. Either you forgot to SAVE the program INVACCTN or you executed your SAVE command improperly and the computer did not SAVE the program and did not tell you otherwise. The first possibility is the most probable.

Number two, the program (especially if you got it from someone else, already on a mini-disk) is too large to RUN on your computer. You either need more memory, a smaller program, or both.

10

HOW TO SAVE A PROGRAM THAT YOU WRITE

Every time you sit down in front of this magic box to write a program for a specific purpose, or a game, or to try some programming technique, the potential to do something great exists, limited only by the human, not by the computer.

On that occasion when you feel that you have accomplished this, which may happen daily, you certainly don't want to deprive mankind of this treasure. If you turn off the computer this jewel will be lost. All that extra cash will disappear, that money you could make if you called it "software" and sold it.

Let's assume that after hours and hours of very trying labor, and just before the point of complete mental fatigue, you finally get a very useful and complex program off and RUNning. It's the greatest thing you ever wrote. For that reason you decide to call it GREATEST. There must be some way to save this valuable asset. There is, and here's how:

1. First, check the program and make sure that all RUNs well and there are no changes you want to make before SAVEing the program.

   On PROMPT or READY type in: LIST

   If you find some changes that you want to make, do it. If you are READY to SAVE the program, then proceed.

2. It is always better to find out exactly how big a program is, rather than guessing. There is a method for telling you exactly how many blocks long a program in the computer is. It involves the use of the built-in FREE (0) function.

   The FREE(0) function tells the programmer how much of the computer's available memory is still FREE to use - how much memory is left. By establishing how much memory you have left after you LOAD in both DOS and BASIC you would do the following:

On PROMPT or READY type in: ! FREE(0)

you get: 10812

This means that without any programs in the computer, other than the operational systems, that I have 10,812 bytes of memory available. This number may vary from system to system, but will remain constant for any given system that is not changed. Once you establish this value for your system - remember it.

Since we already have a program in the computer, and we want to find out how many blocks long it is, we do this:

On READY type in: !(10812 - FREE(0))/256

you get:

25.515625

That means that the size of the program in the computer's memory is 25.515 blocks long. Here's how that was determined:

a. If you typed in: ! FREE(0)

and got : 4280

That means you have 4280 bytes of memory left.

b. If you know how much memory you started with, i.e., 10812 bytes and you know how much you got left, i.e., 4280. Then:

10812 - 4280 = 6532

Which means that you used 6532 bytes to write the program in the computer.

c. If each block contains 256 bytes then:

6532 / 256 = 25.51 blocks

Which means the the program in the computer is 25.51 blocks long.

d. Putting this all together into one expression, you get:

No. of blocks = (10812 - FREE(0))/256

NOTE: See Chapter 36 for an easier way to do this if you have the "new" BASIC.

3. If you are going to SAVE your program on a a brand new mini-disk that has never been used, you must first INITIALIZE the disk. However, if you INITIALIZE your mini-disk at this point you will lose your program in the computer's memory. The INITIALIZing function uses the same portion of memory that your program is in.

   DO NOT INITIALIZE AT THIS POINT

   If you want to SAVE your program at this point you will have to SAVE it to another mini-disk that has already been INITIALIZED.

   In view of the complications brought about by not having an INITIALIZEd mini-disk ready when you need one, it is a good practice to INITIALIZE all brand new mini-disk as soon as you get them. Even if you do several at a time it does not matter, because you can't use them until they are INITIALIZED.

4. We must be in the Disk Operating System - DOS mode - to prepare our FILES. So:

   On PROMPT or READY type in: BYE

5. We must CReate a FILE named GREATEST which is 26 blocks long.

On * type in: CR GREATEST 26

6. We must TYpe the FILE. Since it is a program written in BASIC that can be LOADed and SAVEd, it is a TYpe 2 program. So that the computer knows that, we:

On * type in: TY GREATEST 2

7. The program is in the computer, we have CReated and TYped the FILE, everything is now READY to SAVE the program GREATEST. We must go back to the BASIC mode by:

On * type in: JP 2A04

8. Let's now SAVE the program GREATEST to the mini-disk. We do that by:

On READY type in: SAVE GREATEST

9. It's done. You now have your program named GREATEST SAVEd on a mini-disk. To check it, tell the computer to LOAD it into the computer.

On READY type in: LOAD GREATEST

If READY pops up as the last item on the monitor all went well and you did in fact SAVE your program. If you get an ERROR, nothing is lost, the original program is still in the computer. Just start over with step number 1, but this time change the name of your program to GREAT. This will preclude the necessity of DELETEing FILEs at this time.

And that's all there is to that....

11

HOW TO PUT DOS ON A MINI-DISK

There are certain programs which are more important than others. Some programs are not worth the effort to SAVE and others will literally shut you down if the mini-disk they are stored on is lost or damaged.

One such program of the latter type is your Disk Operating System (DOS). If you lose or damage this program you are out two ways. Number one, you can't do anything with your computer other than use it as a "television typewriter." Without some means of getting BASIC into the memory of the computer, you can't program or RUN in BASIC. Without some means to tell the computer when and how to use the disk-reader you can't LOAD, READ, WRITE, SAVE, LIst, or much of anything else.

The second way you are out is the cost of another mini- disk with the Disk Operating System (DOS) on it. I can assure you that the cost of a blank mini-disk is much, much less expensive than another pre-written DOS programmed mini-disk.

Since we are only human, and a known human trait is the "expectability of screwing up," one can without hesitation be assured that one will at sometime screw up one's DOS disk. So in the true spirit of reality --- Prepare for the worst and expect it. So let's prepare a second, third, or however many additional mini-disk with DOS that your past life experiences comfortably dictate.

Here's how:

1. Place mini-disk with Disk Operating System - DOS in the disk-reader.

   On > type in: EX E900

   Then press RETURN key to enter the command into the computer - as you will after each "type in:" When the * character comes on the screen it indicates that DOS has been LOADed into that part of the computer's memory which operates the disk-reader. You are in the DOS mode.

2. Put your "new" mini-disk into the disk-reader.

   I put quotation marks around "new" for two reasons. First, it doesn't have to be a brand new disk; it can be a mini-disk that you have already used, but no longer need to keep what is SAVEd on it. We are going to INITIALIZE the "new" mini-disk. This procedure will format the recording area so that it will accept information from the system. It will also over-write (erase) anything that is on the mini-disk.

   The second reason for the quotation marks is to to help keep up with which mini-disk is which. Remember that the "new" mini-disk will be the one that we are putting DOS on.

   On * type in: IN

   If you do not INITALIZE a new mini-disk and you try to record on it you will always get a HARD DISK ERROR.

3. Now we must CReate a file named DOS ten blocks long

   On * type in: CR DOS 10

4. Now we must put DOS into the computer a second time, but not as part of the operations system program (as above). This time we we will LOAD it as a "regular program." Since it is just a "regular program" it will be stored in that portion of the computer's memory which can be LOADed and SAVEd by the appropriate command.

   We actually will have DOS in the memory of the computer twice. Once in the operational systems portion of the computer's memory and once in the "regular program" portion of memory.

   To get DOS into the "regular" memory of the computer we must:

On * type in: LF DOS 3000

Which means, LOAD a File named DOS into the computer's "regular" memory whose location (ADDRESS) is 3000. Note that the LOAD command in the DOS mode is different from the LOAD command in the BASIC mode. Since both modes have common commands the names have been changed so as not to confuse the operator. Here are some of the command names:

DOS		BASIC	
LF		LOAD	
LI		LIST	
DE		DEL	(delete)
GO		LOAD	
SF		SAVE	
RD		READ	
WR		WRITE	
JP 2A04	<--->	BYE	

5. We have now LOADed into the computer DOS in two places, INITIALIZED the "new" mini-disk, CReated a file for DOS on the "new" mini-disk, and all that is left to do is SAVE DOS on the newly CReated File. So we put the "new" mini-disk" back in the disk-reader and:

On * type in: SF DOS 3000

After pressing the RETURN key to enter our command, the disk-reader will come on, run for a few seconds, turn itself off, and the character * will appear as the last item on our monitor. This would indicate that all went well, and that DOS is now on the "new" mini-disk.

6. To check to make sure that all went well:

On * press at the same time:

UPPER CASE key and REPEAT key

7. You are back in the "start up" mode.

On > type in: EX E900

If you get a * you just LOADed your copy of the Disk Operating System - DOS.

You not only have now fixed a second DOS disk, you have protected yourself from the effects of an eventual dumb mistake. That's all there is to know about doing that....

12

HOW TO PUT BASIC ON A MINI-DISK

Equally important as knowing how to put DOS on a mini-disk is how to put BASIC on a mini-disk, for all the same reasons. If you were to lose or damage your only copy of BASIC you still couldn't RUN a program other than Disk Operating System procedures. That by itself is not much fun. So, let's find out how to transfer our BASIC language program from one mini-disk to another. The methods are essentially the same as for DOS except the "words" are different.

Here's how:

1. We must first get the computer in the DOS mode.

   From start-up: on > type in: EX E900

   From BASIC: on READY type in: BYE

   Press RETURN key to enter command into the computer. The * character should appear as the last item displayed on the monitor.

2. If your mini-disk is a brand new disk and has never been INITIALIZED before, or it's an old, used mini-disk that you want to completely erase then:

   On * type in: IN

   Press RETURN key to execute the INITIALIZE command.

   If you are adding BASIC to a previously used mini-disk which already has DOS or some other programs on it that you want to keep ---

   DO NOT INITIALIZE

3. Put the "new" previously INITIALIZED mini-disk into the disk-reader. We will first CReate a file named BASIC, forty-five Blocks long, which is the size of our BASIC Language Program.

   On * type in: CR BASIC 45

   (With a single space between each "word")

4. Since our BASIC program is a GO TYpe program. That is, that it is LOADed by the command --- GO BASIC, we must tell the computer this information. We do that by TYpeing the program. There are four TYpes of programs that we will be concerned with:

   TYpe 0 . . A machine language program such as our Disk Operating System program that is LOADed by the EXecute command.

   EX E900

   TYpe 1 . . An operations systems program such as our BASIC Language Program, which is LOADed by the GO command.

   GO BASIC

   TYpe 2 . . A BASIC program such as those that we have written which are LOADed by the LOAD command.

   LOAD INVACCTN

   TYpe 3 . . A data file which can only be accessed by a READ or WRITE statement after it is OPENed, and can not be LOADed or SAVEd.

```
10 OPEN #0, INVACCTN
20 READ #0, A,B,C
```

To tell the computer what TYpe program we have CReated a file for we do the following:

On * type in: TY BASIC 1 2A00

The "2A00" is the location (ADDRESS) to which this TYpe one File is to be stored in the computers memory.

5. Now we must LOAD the BASIC program File into the memory of the computer. We do that by putting a mini-disk with BASIC already on it, the "old" disk, into the disk-reader and:

   On * type in: LF BASIC 2A00

   Which tells the computer to LOAD from the mini-disk a File named BASIC into the memory of the computer's memory at ADDRESS "2A00".

   If after we press the RETURN key, an * is the last item on the monitor, we know that the program is in the computer.

6. We now put the "new" mini-disk into the disk-reader, since that's where the File is that we want to SAVE the BASIC program to. Then we do it by:

   On * type in: SF BASIC 2A00

   Which tells the computer to "record" on the mini-disk whatever is in its memory at location (ADDRESS) 2A00 and identify that File by the name BASIC.

7. You now have BASIC on a second mini-disk. If you don't believe it:

On * type in: GO BASIC

If you don't get back -- READY -- you messed up. Go back to square one.

13

HOW TO DELETE A PROGRAM OR A FILE

It seems that no sooner than you spend untold hours creating a program or a file, you then come up with "a better mouse trap." You no longer have any use for the old one, but you decide to keep it for sentimental value. After all, it was the "first program of that type that I wrote." Soon you have more "old mouse trap" programs SAVEd on "mini-disks" than you have "active" programs.

Then the time comes when you are right in the middle of "a better mouse trap," it's the greatest one you've ever done. You look for a "mini-disk" with enough open blocks (unused space) on it to SAVE this wonderful new program, only to discover that they are all full. You don't hesitate for a minute to grab one of those "old sentimental" ones and erase it. After the first one it becomes easy to reclaim all the rest.

The same thing holds true when you are writing a program. After spending hours putting LINEs in, you figure a better way to do something and you spend micro-seconds getting rid of those same LINEs.

Now let's consider deletion or erasing methods. They are best explained by example. I found that it was harder trying to demonstrate how to get rid of something you didn't have, than to create an example and then get rid of it.

Our first example is that we have a program in the computer which is 50 LINEs and we want to get rid of the whole thing. What we do is SCRatch the entire program. Here's how:

1. On prompt or READY type in: SCR
2. Press the RETURN key.

You have now wiped out all 50 LINEs of a program that was in the computer. If you were to do a LIST, all you would get back is a READY.

Example #2

We have a program in the computer which has 90 LINEs, numbered from 10 to 900, in a 10X series - i.e. 10,20,30,--- 880,890,900 We want to get rid (delete) of LINE NUMBERs 50,290,and 470.

All we have to do is:

1. On PROMPT or READY type in: 50
2. Press the RETURN key.
3. On PROMPT type in: 290
4. Press the RETURN key.
5. On PROMPT type in: 470
6. Press the RETURN key.

These lines are now deleted. If we did a LIST of the program, you would see that they are no longer part of the program. Because it is so easy to delete a LINE, one must take extra caution not to accidentally type in a random number while the computer is in BASIC mode with a program LOADed. If it should happen, do not compound the mistake by pressing the RETURN key. Correct the error by a "shift-delete" or press the " @ " key.

Another common mistake is while you are busy thinking about the content of a LINE, you type in the wrong LINE NUMBER. If you typed in 360 and should have typed in 560 and then caught your mistake and pressed the RETURN key to "start over," everything would seem O.K., but it isn't. You just DELETEd LINE NUMBER 360. Remember, the computer does what you tell it to do --- even when you don't mean it. The proper way to have handled the error would have been a "shift-delete" or to press the " @ " key.

Using the same program as in Example #2, let's now DELETE a larger portion of the program. This time we want to DELETE everything from LINE NUMBER 470 to LINE NUMBER 780. We could do it "by the numbers" as we did in the second example. That would take a long time and would be boring, but it would work. Or we could do it this way:

1. On PROMPT or READY type in: DEL 470,780
2. Press the RETURN key.

The task is now complete.

If you want to verify it type in: LIST 460,790

you get:

```
460 IF K = COS(T) THEN GOSUB 870
790 INPUT "SUM OF ALL FIELDS : ",G7
```

(or whatever the content of these LINEs may have been)

If you wanted to DELETE everything in our Example #2 program from LINE NUMBER 720 to the end of the program you would then:

1. On PROMPT or READY type in: DEL 720
2. Press the RETURN key.

Consider it done.

To confirm it:

On READY type in: LIST

The entire program will scroll by from top to the last LINE, which will be LINE 710.

NOTE: You should be aware that on the system that I am using that due to a flaw in the BASIC, when you want to use those LINE NUMBERS that you earlier DELeted --- all will seem rosy, but it's not.

What happens when you start filling the vacuum that you created by DELeting that big block of LINEs, and you keep adding one LINE after another, and there's no indication that anything has gone awry, is that all these efforts are in vain. Even though everything you type appears on the monitor, it's continuing to be DELeted by the computer from the program. Consequently you are not doing anything except practicing typing.

Knowing this still makes the DELete function a usable programming tool. Not knowing this renders the thing less than useless. You can correct this flaw by doing the following each time you DELete a large block of LINEs:

1. After you have done the DEL 470,780 and pressed the RETURN key. on READY type in: BYE
2. Press the RETURN key.
3. On * type in: JP 2A04
4. Press the RETURN key.

and you are READY to add those LINEs.

All we did was leave BASIC mode, which had the effect of wiping out any remaining "commands," by going to DOS mode. Then we JumPed back in BASIC mode with a "clean slate."

You are now in a position to wipe out a LINE, a portion of a program, or the entire program. We have gone far beyond just "scratching the surface" with regard to the subject of DELetions, we have ventured into realm of "SCRatching the whole thing."

Well, not exactly. If I had decided to write an entire chapter on the subject of how to DElete a FILE on a "mini-disk," it would end up a short,three line chapter and still cover the subject completely, almost, like this:

1. On * type in: DE NAME
2. Press the RETURN key.

For NAME substitute the name of the file you wish to DElete.

And that covers the subject of program and file DELETE.

14

HOW TO DELETE A FILE FROM A DISK

Let's say that you develop the good habit of saving every significant program that you write. Even when you write another program that does the same thing quicker and better you hang on to the original. At some point in time you will find that many of your earlier programs, though great at the time, are no longer worth keeping. You find that all those mini-disks represent too large of an investment to sit around with no expectation of being used again. It's time to clean house.

If you have a mini-disk with several programs on it and you deem that none are worth keeping, and you want to erase the entire mini-disk, here's how:

1. Put the computer in DOS mode.

   a. From start up type in: EX E900

   b. From BASIC on READY type in: BYE

2. Put the mini-disk to be cleared in the disk-reader

3. When you INITIALIZE a mini-disk the computer over-writes everything that is on the mini-disk as it formats it to accept data from the computer This has the same effect as "erasing" the mini-disk.

   Here's how:

   On * type in: IN

   Note that when you INITIALIZE a mini-disk and you get to the DOS mode from BASIC mode (BYE), you can not JumP (JP 2A04) back to BASIC mode. Both BASIC and any program that you have in the computer's "regular" memory (programs that you write or LOAD) are erased as well. To get back to BASIC you must GO BASIC.

Now let's suppose that you don't want to "clean" the entire mini-disk, but only remove one FILE. The name of the FILE that we want to DElete is GONE.

Here's how:

A. Put the computer in the DOS mode as above.

B. Place the mini-disk with the FILE named GONE in the disk-reader.

C. On * type in: DE GONE

When the * reappears on the monitor as the last item - GONE is gone, it has been DEleted.

D. To check it, you can LIst the FILEs to get the names of all the FILEs on the mini-disk by:

On * type in: LI

You LIsting should not contain the FILE named GONE.

So that you are aware of what really happens when you DElete a FILE, consider how the FILE Directory is set up. The first four blocks of a mini-disk is reserved for the FILE Directory. When you issue a command regarding a FILE name, the computer tries to match the name you ask for with the names in the FILE Directory. If it is successful, the computer then reads the Disk Address of that FILE, which is recorded as part of the FILE Directory, and goes to that Disk Address and executes the command that you gave it.

When you DElete a FILE the computer doesn't actually erase the FILE, but only the Disk Address. The FILE is left intact. If you were to CReate another FILE with the exact same Disk Address you would be able to access that "DEleted" FILE.

This method is actually used to change the name of a FILE without changing the file. Consider this example:

You do a LIst of a mini-disk and it tells you:

GONE 186 23 2

There is a FILE with the name GONE at Disk Address 186 that is 23 blocks long and is a TYpe 2 program FILE (can be LOADed and SAVEd).

If you want to change the name from GONE to HERE and still maintain the FILE you do this:

a. On * type in: DE GONE
b. On * type in: CR HERE 23
c. On * type in: TY HERE 2

You now have the same FILE with a "new" name. If you LIsted it you would get:

On * type in: LI

you get:

HERE 186 23 2

Which is exactly what you wanted, an old FILE with a new name, and the same Disk Address.

Now what happens when you CReate a new FILE and it ends up with the same Disk Address as a DEleted FILE, but you really want a "new" FILE and not the old recorded information still on the mini-disk from the DEleted FILE, usually nothing.

On occasion, if you WRITE to the "new" FILE in the exact same format as you did the "old" FILE and you READ from the "new" FILE in the same format as you did from the "old" FILE, you will find that you are using "old" FILE data if it's a TYpe 3 FILE. You might end up LOADing an "old" program if you did not SAVE the new program to the "new" FILE.

To prevent this from happening with a TYpe 3 data FILE, all you have to do is be sure that the "new" data starts WRITEing at the start of the "new" FILE and not at the end of

previous data for the "old" DEleted FILE. This is done with the TYPE statements (as opposed to TYpe) used in conjunction with the READ statement in your program.

This presents no real problem in using TYpe 3 FILEs if you are aware of how the FILE Directory and the FILEs are actually used by the computer.

If you DElete several FILEs from a mini-disk and leave several other scattered in between those DEleted, ratner than lose this available "space" or hope to have a program of the proper length (no. of blocks) to fit in, you can COmpress all the remaining FILEs to the front of the mini-disk. This will leave all of the available "space" at the end of existing FILEs.

To do this you:

1. The computer must be in the DOS mode.

   a. From start up on > type in: EX E900
   b. From BASIC on * type in: BYE

2. Place mini-disk to be COmpressed into the disk-reader.

3. On * type in: CO

   When you press the RETURN key to enter your command into the computer, you will note that the disk-reader will come on and run longer than usual, with lots of "clicks."

   What it is doing is ReaDing each program into the computer's holding area memory (buffer) until it locates or clears sufficient "space" to WRite it back on the mini-disk at the "front" of the disk. It will do this with each program on the mini-disk and the automatically change the Disk Address to match the programs new location.

That's all there is to that....

15

HOW TO COPY A BASIC PROGRAM FROM ONE MINI-DISK TO ANOTHER

Nobody should ever keep only one copy of an important or highly used BASIC program. There should always be one to use and one or more to store. The out of pocket cost for the extra mini-disk is far less than the cost in time and effort to rewrite a program you've already done. Besides that, it's far less interesting rewriting a old program than it is to create a new one.

Then there's the cost of BASIC software. (It's called software if you buy it.) A pre-programmed mini-disk with some utility type program or a game program that is written in BASIC is generally to expensive to risk loss or damage, when it can be duplicated for just a dollar or so.

The average cost of a mini-disk is about five dollars. Each mini-disk will hold 89,600 characters (bytes). That works out to about 5.6 cents per 1000 bytes (1 K), and that's cheap.

Each mini-disk consists of 35 tracks, each track contains 10 blocks, and each block contains 256 bytes. Not all of these blocks can be used for recording programs. The first four blocks are reserved for the directory of the contents of the remaining 346 blocks. When you LIst a mini-disk it is from these first four blocks that that information is obtained. Here is a typical LIsting:

```
DOS        4  10  0
BASIC     14  45  1  2A00
FILE*L    59   5  2
NUMBERS   64  10  2
6/10/78   74   5  2
MT-DATA   79 100  3
$$$$$$$  179  16  2
```

The first eight spaces are for the FILE name, next is the starting Disk Address, then the size of the FILE - number of blocks, then the FILE TYpe. The BASIC Language program FILE has an additional computer memory Address which tells the computer exactly where to put it - in the computer's memory.

If you will note that the starting Disk Address for each consecutive FILE is the sum of the previous FILE plus the total number of blocks used by that FILE. Thus the FILE named

NUMBERS, in our example above, has a starting Disk Address of 64, and it is 10 blocks long, so the next FILE's starting Disk Address will be 74, and it is.

When the computer is told to LOAD a FILE it does not search the entire mini-disk for that FILE, but reads just the FILE directory - the first four blocks on the mini-disk. If it finds the FILE name that it is searching for, it then reads the Disk Address and the size - number of blocks, which tells it where to go on the mini-disk to LOAD the FILE. That is, after it checks the last item on the line and makes sure that the type of FILE you are sending it after is the right TYpe.

Now, knowing all of this information should make it easy to LOAD a program from one mini-disk and record it on another. The fact of the matter is that even without knowing all of this it's relatively easy to do. Here's how:

1. Put the computer in the Disk Operating System - DOS mode.

   From start up put a mini-disk with DOS in the disk-reader and:

   On > type in: EX E900

   From BASIC mode:

   On READY type in: BYE

2. If you are using a brand new mini-disk to put your program on or you are using an used mini-disk that you want to totally erase, you must first INITIALIZE the mini-disk.

   To do this you:

   On * type in: IN

   If you are going to add this program to a mini-disk that already has been INITIALIZED do not do it again -- skip this step.

3. Put the mini-disk with the program on it that you want to record on the "new" mini-disk, in the disk-reader. We are going to LIst it to get the exact "spelling" of the FILE name that we want to record and also to get the size - number of blocks it contains, and the TYpe

   To do this:

   On * type in: LI

   We must note the following items for later use. We will use the above sample LIsting for our example.

   The FILE name: NUMBERS
   The FILE size: 10 blocks
   The FILE TYpe: 2

4. Put the "new" mini-disk back into the disk-reader We must CReate a FILE with the name NUMBERS, 10 blocks long.

   To do that we:

   On * type in: CR NUMBERS 10

5. Now we must tell the computer what TYpe of FILE NUMBERS is so that it will know how to access it.

   We do that by:

   On * type in: TY NUMBERS 2

   NUMBERS is now TYped as a program written in BASIC which can be LOADed and SAVED.

6. Take the "new" mini-disk out of the disk-reader and put in the mini-disk with the program on it that you want to record. We must now go to the BASIC mode.

   Do that by:

   On * type in: GO BASIC

7. We now will LOAD the program named NUMBERS into the computer by:

   On READY type in: LOAD NUMBERS

8. Take out that mini-disk and put in the "new" mini-disk that we want to SAVE NUMBERS on.

   To do that:

   On READY type in: SAVE NUMBERS

   You now have two copies of NUMBERS.

9. If you do not have "complete faith" and want to see.

   Do the following:

   a. On READY type in: SCR

   This will SCRatch the program in memory.

   b. On READY type in: LIST

   This will LIST any program in memory
   Since we just SCRatched it, this shows
   you that there is nothing there.

   c. On READY type in: LOAD NUMBERS

   This will put NUMBERS into memory from your "new" mini-disk.

   d. On READY type in: LIST

   The computer will scroll by the program that you copied.

And that's all there is to that.....

16

THE LOOP

The LOOP is exactly what it sounds like it is, a course that one follows that leads him in circles. Actually it's not us that will go in circles, it's the computer. The "simplest" LOOP that I can think of is:

```
10 GOTO 20
20 GOTO 10
```

As you may have noticed, I put "simplest" in quotation marks, as the word has several definitions.

Program LOOPS generally instruct the computer to do the same thing over and over a specified number of times. They are best explained by example.

Consider this program:

```
10  FOR K = 1 TO 3
20  PRINT K
30  NEXT K
```

If we RUN we get:

```
1
2
3
```

This is called a FOR LOOP. LINE 10 tells the computer to execute the program three times. Each time it executes the program it is to assign a new value to the variable K. The specified values for K are 1,2, and 3. The computer will automatically increase the value of the variable by increments of 1, between the limits set in the FOR statement, and then execute the program with the new value.

Another type of LOOP which will do the same thing, only different, is the GOTO LOOP.

Consider this:

```
10  R = R + 1
20  IF R = 4 THEN END
30  PRINT R
40  GOTO 10
```

The computer can be placed in a LOOP by many different programming techniques. Usually these different methods are the choice of the programmer rather than being the result of a specific type LOOP for a specific type technique. The most discussed and the most used LOOP is the FOR LOOP.

Consider this program:

```
10  FOR T = 1 TO 3
20  X = 10
30  PRINT X + Y
40  Y = Y + X
50  NEXT T
```

In this FOR LOOP the values of T are not actually involved in the essence of the program, but the LOOP is just a method to make the program cycle three times to change the value of the algebraic expression -- X+Y. If we were to change LINE 10 to read:

```
10 FOR T = 1 TO J
```

Where the value of J is determined by an INPUT statement, information to be typed in, or as a result of a calulation made by the computer in some other part of the program, then the LOOP will cycle through the algebraic expression J times.

The computer will automatically increase the value of the variable in a FOR statement by 1 unless you tell it to do otherwise. If you want the computer to increase the value of the variable in a FOR statement by increments of .5 rather than by 1, then you must tell the computer to STEP the increases by .5 .

This is how:

```
10  FOR  X  =  1  TO  3  STEP .5
20  PRINT X,
30  NEXT X
```

If you RUN you get:

```
1  1.5  2  2.5  3
```

You can STEP a FOR statement by any size increment that falls within the bounds of the statement. We could have STEPed the above LINE 10 by .00063, 1.87629, or 2.99. We could not have STEPed it by a minus number, a zero, or a number greater than three.

The computer will not function with a FOR statement which goes from a higher number to a lower one.

Such as:

```
10  FOR  X  =  3  TO  1
```

However, it will work if the "negative" FOR statement is STEPed. Why this happens is beyond the scope of this book. It is also beyond the scope of its author.

But:

```
10  FOR  X = 3 TO 1  STEP  -.5
20  PRINT X,
30  NEXT X
```

If we RUN we get:

```
3  2.5  2  1.5  1  .5
```

A program can have as many FOR LOOPS as the programmer desires. However, if they are superimposed upon each other, rather than being separate and complete LOOPS isolated in different parts of the program, they must be placed in the program in a specified manner.

Consider this program:

```
 10  FOR  X  =  1  TO  7
 20  FOR  Y  =  4  TO  7
 30  FOR  Z  =  8  TO 11
 40  FOR  W  =  11 TO  14
 50  N = X + Y + Z + W
 60  PRINT N ,
 70  NEXT W
 80  NEXT Z
 90  NEXT Y
100  NEXT X
```

If we RUN we get:

```
24 25 26 27 25 26 27 28 .....35 36 37 38 36 37 38 39
```

Note that the FOR LOOPS are "fed" from the innermost LOOP to the outermost LOOP. These are called "nested" LOOPS. You must always be sure that you follow this method when you put in your NEXT statements. Any time you have a FOR statement, you must also have a NEXT statement. If you have six FOR statements in a program, you must also have six NEXT statements in that program.

This is essentially all that you need to know about LOOPs to get started using them......

17

THE SUBROUTINE

This is not a chorus line of underwater ships, but a method of telling the computer to do something numerous times without having to write out the instructions numerous times. Being able to use the GOSUB Statement keeps the programmer from being bored by not having to write and then rewrite, and then rewrite, and then rewrite . . . . a portion of a program that is repeatedly used. All he or she has to do is write that portion of the program once, and then anytime it is needed, you just tell the computer to go to it.

Consider the following program which is used to calculate the discount of different types of inventory items and different quantities. This program demonstrates the use of the GOSUB statement and the RETURN statement.

```
 10 INPUT "HOW MANY OF CLASS #1 ITEMS : ",A
 20 GOSUB 180
 30 !"THE DISCOUNT IS  :  ",B*100," %"
 40 !
 50 INPUT "HOW MANY OF CLASS #2 ITEMS : ",A
 60 GOSUB 180
 70 !"THE DISCOUNT IS  :  ",B*93," %"
 80 !
 90 INPUT "HOW MANY OF CLASS #3 ITEMS : ",A
100 GOSUB 180
110 !"THE DISCOUNT IS  :  ",B*75," %"
120 !
130 INPUT "HOW MANY OF CLASS #4 ITEMS : ",A
140 GOSUB 180
150 !"THE DISCOUNT IS  :  ",B*25," %"
160 !
170 GOTO 10
180 IF A > 10 AND A < 100  THEN B = .07
190 IF A > 99 AND A < 500  THEN B = .09
200 IF A > 499 AND A < 1000 THEN B = .12
210 IF A > 1000 THEN B = .15
220 RETURN
```

Now let's consider what is happening. First the computer asks you, "HOW MANY OF CLASS #1 ITEMS"? You type in a response. It then assigns that value to the variable A. Then the computer jumps to a subroutine starting at LINE 180. This is initiated by the GOSUB statement:

20 GOSUB 180

The computer then evaluates the value of A, quantity of CLASS #1 items, through LINE 180 to LINE 210. It then assigns a value to the variable B, based on its evaluation of variable A. (Note that the greater the number of items, the greater the discount factor.)

After the computer determines the "discount factor" it sends this information back up to the regular program. The interrupted regular program picks up at the next statement after the GOSUB statement and continues to execute the rest of the program in a normal manner. To tell the computer to do all of this, i.e., to RETURN to the regular program, one LINE past where it left, your LINE statement would be:

220 RETURN

At this point the computer determines the actual discount by multiplying the discount factor by a "class factor." The computer is then instructed to PRINT (!) the results of its efforts and go back for more.

If we RUN we get:

```
HOW MANY OF CLASS #1 ITEMS : we type in: 375
THE DISCOUNT IS 9 %

HOW MANY OF CLASS #2 ITEMS : we type in: 20
THE DISCOUNT IS 6.51 %

HOW MANY OF CLASS #3 ITEMS : we type in: 1200
THE DISCOUNT IS 11.25 %

HOW MANY OF CLASS #4 ITEMS : we type in: 832
THE DISCOUNT IS 3 %
```

Now let's track what happened:

1. You assign a value to A = 375
2. The computer evaluates A in the subroutine and assigns a value to B = .09, since 375 is greater than 100 and less than 500.

3. **The RETURN statement (LINE 220) send the computer back to where it came from - the next LINE after the GOSUB statement (LINE 30).**

4. **The computer then determines the discount:**

   B * 100
   .09 X 100 = 9

   and PRINTs the discount.

5. **Then as expected the computer moves on to the next LINE in the program (LINE 40).**

The GOSUB statement is no more than a fancy GOTO statement with a "leash" to bring it back after it done its business, a RETURN function. The entire activity of the GOSUB statement could be replaced with two or more GOTO statements. One to send the execution of the program to another LINE and one to send it back. The main advantage is that you don't have to keep up with RETURN LINE NUMBERS, the computer does it for you.

If you understand the above program, then you understand the GOSUB statement. If you don't, go back and study it, it's really simple.....

18

THE "ON" STATEMENT --- WHAT IT IS AND HOW TO USE IT

Unlike most of the other topics thus far discussed, the "ON" STATEMENT is not very complicated nor does it need very much explanation. This is not an indication of its lack of importance but of its simplicity. Although the uses of the ON statement are varied, no matter where it is used, it always does the same thing --- transfers control of the program to another LINE depending on the value of a specified NUMERIC variable. That's "computer people" talk for a multiple choice GOTO statement.

A typical ON statement in a program would look like this:

```
40  ON  C  GOTO  70,110,270
```

What this tells the computer is:

```
IF C = 1  THEN GO TO LINE 70
IF C = 2  THEN GO TO LINE 110
IF C = 3  THEN GO TO LINE 270
```

The value of C can be "typed in" as a response to an INPUT statement, can be the result of a mathematical expression, or can simply be assigned by a LET statement, i.e., 30 LET C = 5. A useless program which illustrates how the ON statement works is this:

```
 1!1
 2!2
 3!3
 4!4
 5!5
 6!6
 7!7
 8!8
 9!9
20 INPUT "TYPE A NUMBER FROM 1 TO 10 : ", H
30 ON H GOTO 1,2,3,4,5,6,7,8,9,40
40 END
```

If we RUN 20 we get:

```
TYPE A NUMBER FROM 1 TO 10 : if we type in 6
6
7
8
9
TYPE A NUMBER FROM 1 TO 10 : if we type in 8
8
9
TYPE A NUMBER FROM 1 TO 10 : if we type in 10
READY
```

What is happening is that the number we type in is evaluated by the ON statement, and the computer is being sent to that LINE to execute the program. The computer goes to the LINE number equal to the INPUT value of H, and continues to execute the rest of the program from that point on. When you type in the number 8 for the INPUT statement, it gives the value of 8 to the variable H. The computer will then GOTO the eighth LINE number in your ON statement and the program will proceed from that LINE.

I do not know the upper limit for the values of the variable in an ON statement, but I do know that it can be up to 24. Branching to more LINEs than this would require branching to another ON statement, if the need should arise.

You may have noticed that I said RUN 20. This has nothing to do with the ON statement, but just starts the execution of the program at LINE 20, the INPUT statement, rather than at LINE 1.

Another program which illustrates a use of the ON statement is the following:

```
10 !"  WHEN YOU MIX TWO DIFFERENT PRIMARY COLORS  "
20 !"  YOU GET A THIRD COLOR .     "
30 !
40 !" SELECT THE COLORS TO BE MIXED BY NUMBER "
50 !
55 ! "          COLOR  #1   BLUE  "
60 ! "          COLOR  #2   YELLOW  "
70 ! "          COLOR  #3   RED  "
80 !
90 INPUT "WHAT IS YOUR FIRST COLOR : ", C
100 INPUT "WHAT IS YOUR SECOND COLOR : ", D
```

```
110 !
120 ON C GOTO 130,200,270
130 ON D GOTO 140,160,180
140 !"        BLUE  +  BLUE  =  BLUE     "
150 GOTO 340
160 !"        BLUE  +  YELLOW  =  GREEN  "
170 GOTO 340
180 !"        BLUE  +  RED  =  PURPLE  "
190 GOTO 340
200 ON D GOTO 210,230,250
210 !"        YELLOW  +  BLUE  =  GREEN  "
220 GOTO 340
230 !"        YELLOW  +  YELLOW  =  YELLOW "
240 GOTO 340
250 !"        YELLOW  +  RED  =  ORANGE "
260 GOTO 340
270 ON D GOTO 280,300,320
280 !"        RED  +  BLUE  =  PURPLE "
290 GOTO 340
300 !"        RED  +  YELLOW  =  ORANGE "
310 GOTO 340
320 !"        RED  +  RED  =  RED  "
330 !
340 !
350 !"   DO YOU WANT TO CONTINUE ????????? "
360 INPUT "TYPE IN :  1 for YES  &  2 for NO > ", E
370 ON E GOTO 40,380
380 END
```

If we RUN we get:

```
WHEN YOU MIX TWO DIFFERENT PRIMARY COLORS
YOU GET A THIRD COLOR .

SELECT THE COLORS TO BE MIXED BY NUMBER

        COLOR #1  BLUE
        COLOR #2  YELLOW
        COLOR #3  RED

WHAT IS YOUR FIRST COLOR :  we type in:  1
WHAT IS YOUR SECOND COLOR : we type in:  3

         BLUE  +  RED  =  PURPLE

DO YOU WANT TO CONTINUE ?????????

TYPE IN  1 for YES  &  2  for  NO  :  we type in: 2
READY
```

Now, let's examine what took place. When we typed in 1 for the first INPUT statement (first color ?), we assigned the value of 1 to the variable C. When we typed in 3 for the second INPUT statement (second color ?), we assigned the value of 3 to the variable D.

```
120  ON  C  GOTO  130,200,270
```

When the computer got to LINE 120 the ON statement sent the continued execution of the program to LINE 130, since the value of C was equal to 1. Had C been equal to 2, the computer would have been sent to LINE 200; the second LINE number in the ON statement. C equal to 3 would have sent it to LINE 270.

LINE 130 is another ON statement. For our second INPUT statement we typed in 3. This assigned the value of 3 to the variable D.

```
130  ON  D  GOTO  140,160,180
```

When the computer executed LINE 130 with the INPUT value of 3 for D, it progressed to LINE 180; the third LINE number in the ON statement.

```
180  !"    BLUE  +  RED  =  PURPLE  "
```

This PRINT statement yields:

```
BLUE  +  RED  =  PURPLE
```

Then we:

```
190  GOTO  340

340  !
```

All this does is skip a space in our OUTPUT. This is really a PRINT statement, and it actually does PRINT a line of "spaces."

```
350 ! " DO YOU WANT TO CONTINUE ????? "
```

Another PRINT statement which PRINTs whatever you have enclosed in quotation marks.

```
360  INPUT " TYPE IN  1 for YES  &  2  for  NO : ", E
```

Which gives us:

```
TYPE IN  1 for YES  &  2 for NO : we typed in  2
```

This INPUT statement assigned a value of 2 to the variable E. Then the program progressed to the next LINE.

```
320  ON  E  GOTO  40,390
```

Depending on the value of E, this LINE would either send the execution of the program back to the start -- LINE 40 if E were equal to 1, or to the END -- LINE 390, if E is equal to 2. We chose the latter.

And that's where we are with the ON statement -- the end.....

19

THE BUILT IN TABULATOR --- TAB

The next built in function, which is part of most BASIC language programs, that I feel important enough to detail, is the TAB FUNCTION. The dictionary defines "tabulate" as "To arrange in an orderly manner." That is exactly what the TAB function will allow you to do with your OUTPUT --- PRINTed or displayed information.

The TAB function allows you to PRINT out any data in almost any format that you choose. The TAB function must always be used within a PRINT statement, because it tells the computer where to PRINT the material in that PRINT statement. A typical PRINT statement in a BASIC program, used with the TAB function would appear like this:

```
130 ! TAB(30),S
```

This tells the computer to PRINT the value of the NUMERIC variable S thirty (30) spaces from the left margin, or to skip 29 spaces and then PRINT the value of S.

You might have the need of setting up columns of numbers, such that you PRINT statement might look like this:

```
130 ! TAB(10),A,TAB(20),B,TAB(30),C,TAB(40),D
```

This PRINT statement would start PRINTing A at "space" 10, B at "space" 20, C at "space" 30, and D at "space" 40.

As with any PRINT statement, STRING variables (words) can also be used with the TAB function. Consider this LINE:

```
130 ! TAB(10),"WHAT COLOR IS IT ? ",TAB(30),K$,TAB(40),R$
```

If K$ = RED and R$ = BLUE, then this LINE would yield:

```
WHAT COLOR IS IT ?  RED       BLUE
```

Now let's consider a more common use of the TAB function, addressing envelopes. Consider a portion of a mailing list program which prepares the address label as below:

```
70 ! N$,N1$,A$,C$,Z,M$
```

```
Where:    N$  =  customer name
          N1$ =  company name
          A$  =  street address
          C$  =  city & state
          Z   =  zip code
          M$  =  any message
```

```
If:   N$  =  Mrs. Anita Tempanny
      N1$ =  Tootums Nickle Co.
      A$  =  1010 Dime Lane
      C$  =  Salada, Tenn.
      Z   =  10102
      M$  =  Attention Change Dept.
```

If these values were PRINTed by our LINE 70 above we would get:

```
     Mrs.Anita TempannyTootums Nickle Co.1010 Dime Lane      Salada,
Tenn.10102Attention Change Dept
```

Which is no way to address a letter.

First we must spread out our PRINT statement so that each line of the address is to itself.

```
 70 ! N$
 80 ! N1$
 90 ! A$
100 ! C$
110 ! Z
120 !
130 !
140 ! M$
```

If we RUN we get:

```
Mrs. Anita Tempanny
Tootums Nickle Co.
1010 Dime Lane
Salada, Tenn.
10102


Attention Change Dept.
```

Which is still no proper way to address a letter. So we change our PRINT statements to:

```
70  ! TAB(20), N$
80  ! TAB(23), N1$
90  ! TAB(26), A$
100 ! TAB(29), C$ ,TAB(48),Z
110 !
120 !
130 ! TAB(10), M$
```

This will get you:

```
                    Mrs. Anita Tempanny
                       Tootums Nickle Co.
                          1010 Dime Lane
                             Salada, Tenn.    10102

   Attention Change Dept.
```

Which is a properly addressed envelope.

And that's how that works........

20

THE RANDOM FUNCTION - FUN AND GAMES

The RANDOM FUNCTION --- RND(0) --- is very interesting and fun to play with, but I have yet to find a use for it other than in writing games or just watching it generate random numbers. But, there are no rules which say you can't have fun playing with the computer. I highly recommend writing game programs, it's much like reading comic books to learn how to read, it's palatable and it works.

First, I must tell you that the RANDOM FUNCTION dosen't generate random numbers. It generates pseudo-random numbers. The difference is that a random number would be any number selected at random without limits --- the RANDOM FUNCTION can not do this. The RANDOM function - RND(0) - has limits, and within a given program it always produces the same series of random numbers. This may limit its usefulness, but certainly does not negate it.

Let's first examine a program which will give us four random numbers.

```
10  FOR R = 1 TO 4
20  PRINT RND(0),
30  NEXT R
```

If we RUN we get:

```
.15625 .52392578 .21208191 .844149
```

If you were to RUN the same program again, you'd get the exact same series of numbers. This phenomenon provides a method for debugging programs, since you get the same series of numbers each time. To the best of my knowledge, you can not generate truly random numbers, but there are ways to get close.

As you can see in our above program, all the numbers generated by the function RND(0) are decimal numbers. All the numbers are between zero and one. That's all that the RANDOM function does - generate numbers between 0 and 1. Let's say that you want to generate four random numbers between 1 and 10. Since we know exactly what the RANDOM function does, all we have to do is change our program to:

```
10  FOR  R = 1 TO 4
20  PRINT  10*RND(0),
30  NEXT R
```

If we RUN we get:

```
1.5625 5.2392578 2.1208191 8.44149
```

All we have done is to multiply the original random numbers by ten. We now have random numbers between 1 and 10. Let's suppose that we want only whole numbers between 1 and 10. Using essentially the same program, we make the following change:

```
10  FOR  R = 1 TO 4
20  ! INT(10*RND(0)),
30  NEXT R
```

We RUN we get:

```
1 5 2 8
```

Right off, I'm sure that you noticed that I have slipped in a function that has not been mentioned before -- the INTEGER function. The entire activity of this function can be summed up in one sentence. The INTEGER FUNCTION -- INT(K) -- rounds off to the next lowest whole number any value of K. If the value of K = 7.878, the INT(K) = 7. If the value of K = .8453243, the INT(K) = 0. Thus, in our above program all we have done is "round off" the random numbers, once they have been generated and multiplied by ten.

```
20  ! INT(RND(0)*10)

FIRST :   RND(0)        =   .15625
SECOND:   .15625*10     =  1.5625
THIRD :   INT(1.5625) = 1
```

Now let's suppose that we want the computer to PRINT four random numbers between 1 and 60. We would change our program to:

```
10  FOR  R = 1  TO  4
20  PRINT  INT(60*RND(0)+1),
30  NEXT R
```

If we RUN we get:

```
12 48 26 51
```

This is what happened:

```
20 ! INT(60*RND(0)+1)

   FIRST :   RND(0)         =    .19287
   SECOND:   60*.19287      =  11.5722
   THIRD :   11.5722 + 1    =  12.5722
   FOURTH:   INT(12.5722)   =  12
```

That's how the computer got the first random number in the program above.

Note that our range of random numbers from 1 to 60 was established by multiplying the RANDOM function - RND(0) - by the highest number in our desired range and then adding 1 to give us the lowest number in our range.

```
20 ! INT(60*RND(0)+1)
         60   to   1
```

This holds true for all ranges of random numbers as long as the lowest number in the desired range is 1. It would not work if we wanted to generate four random numbers between 300 and 800. If we wrote our PRINT statement like this:

```
20 ! INT(800*RND(0)+300)
         800   to   300
```

We would generate random numbers between 300 and 1100, not 300 and 800. If you will examine the arithmetic involved you will quickly see why. In order to get the desired results our PRINT statement would have to be:

```
20 ! INT(500*RND(0)+300)
```

If we RUN we get:

```
378
709
721
566
```

That concludes most of what there is to know about the RANDOM FUNCTION and how it works. There is one last "secret" which I indicated above that I would reveal, and that is how to make the RANDOM function more random. Even though the computer "helps" us out by generating the same series of "random" numbers each time it executes a given program, when playing games it's not much of a challenge if you know all of the answers beforehand. That is what will happen after you have used the same program several times -- you will end up knowing the series of "random" numbers that the computer is going to generate. It's much more of a challange to write a game program and then be forced to play it without prior knowledge.

If you will look close at the RANDOM FUNCTION's format you will notice a heretofore unmentioned zero enclosed by parentheses. This is the key to an almost real RANDOM number generator. The zero can be replaced by a variable so that:

RND(0) becomes RND(J)

The variable J then becomes a "seed" which will start the RANDOM generator at different "places". It is only used by the computer to "start" the RANDOM generator and need not be gone back to in a given program each time a RANDOM number is generated. However, if you use the same value for the "seed" every time you RUN a given program, you will end up back where you started; getting the same series of "random" numbers. What you must do is to have the "seed," the value for J, be the result of an INPUT statement. Such as:

```
10  INPUT "TYPE IN ANY NUMBER BETWEEN  0  AND 100 : ", P
20  LET  J = -P/100
30  FOR  D = 1  TO  4
40  !  INT(60*RND(J)+20)
50  NEXT  D
60  GOTO 10
```

This program will generate random numbers between 20 and 80. If each time the program comes to the INPUT statement you type in a different number between 0 and 100, you will get a different series of "random" numbers. If you type in the same number you will get the same series of "random" numbers.

The value of the "seed" J can not be greater than the numbers generated by the RANDOM function. It must be between -1 and 1. So if our INPUT statement asks for a number between 0 and 100, we must divide that INPUT by 100 to get a fractional value for the "seed" variable J (J = -P / 100). If any number greater than 99 were typed in, the computer would indicate an OUT OF BOUNDS error. This is because the RANDOM function -- RND(0) -- can not equal or exceed 1.

The value for the "seed" does not have to come from an INPUT statement, it can be the result of any number of devious or complicated routes. The harder it is to purposely duplicate the "seed," the truer will be the "random" number. The only thing that the programmer must be sure of, is that once a value for the "seed" is attained, that it be between -1 and 1.

I have now told you all I know about RND(0) .....

21

THE SIGN FUNCTION

The SIGN FUNCTION - SGN(A) - when called upon, looks at a numeric expression generated by the program that it is used in and assigns itself a numeric value of 1,0,-1, depending on whether the evaluated numeric expression was positive, equal to zero, or negative; respectively.

```
If  K = 58.941  the   SGN(K) = 1
If  K = 0       the   SGN(K) = 0
If  K = - .386  the   SGN(K) = -1
```

Where K is the numeric variable to be evaluated by the SIGN function --- SGN(K)

Although limited in its use, the SIGN function does provide a needed "service." It of course, should never be confused with the SINE function - SIN(G) - as it has nothing to do with that kind of stuff.

Here is a potentially useful program which best illustrates the use of the SIGN FUNCTION:

```
 10  REM  THIS IS A SIMPLE CHECKBOOK BALANCING PROGRAM
 20  INPUT " STARTING BALANCE FROM BANK STATEMENT : ", A
 30  !
 40  INPUT " HAVE YOU MADE A DEPOSIT - YES or NO : ", A$
 50  IF  A$  =  "NO"  THEN  130
 60  !
 70  INPUT " WHAT WAS THE AMOUNT OF THE DEPOSIT : ", E
 80  F = F + E
 90  K = K + 1
100  !
110  INPUT " DID YOU MAKE ANOTHER DEPOSIT : ", A1$
120  IF  A1$  =  "YES"  THEN  70
130  !
140  INPUT "WHAT IS THE AMOUNT OF THIS CHECK  : ", B
150  C = C + 1
160  D = D + B
170  !
180  INPUT "HAVE YOU WRITTEN ANOTHER CHECK ???   : ", Y$
```

```
190  IF  Y$  =  "YES"  THEN  140
200  G = A + F - D
210  !\!\! "*******************************************"
220  !
230  ! "YOU HAVE WRITTEN ",C," CHECKS : TOTAL ",%$C10F2,D
240  ! "YOU HAVE MADE ",K," DEPOSITS  : TOTAL ",%$C10F2,F
250  !
260  H  =  SGN(G)
270  IF  H = 1 THEN 320
280  IF  H = -1 THEN 350
290  ! "      Y O U  A R E  N O T  O V E R D R A W N   "
300  ! "   BUT THERE IS NO MONEY LEFT IN YOUR ACCOUNT  "
310  GOTO 380
320  ! "    Y O U  S T I L L  H A V E  M O N E Y  L E F T "
330  ! "      THERE IS  ",%$C10F2,G,"  IN YOUR ACCOUNT "
340  GOTO 380
350  ! "*****  Y O U  A R E  O V E R D R A W N  ***** "
360  ! "      YOU MUST DEPOSIT  ",%$C10F2,ABS(G)
370  ! "       TO COVER CHECKS ALREADY WRITTEN  "
380  END
```

This program contains several notable programming techniques, most of which have nothing to do with the SIGN function, but that you should be aware of. First we will list the variables contained in the program:

A	=	STARTING BANK BALANCE
A$ and Y$	=	YES or NO
E	=	AMOUNT DEPOSITED
F	=	TOTAL AMOUNT DEPOSITED
K	=	TOTAL NUMBER OF DEPOSITS
B	=	AMOUNT OF CHECK WRITTEN
D	=	TOTAL AMOUNT OF CHECKS WRITTEN
C	=	TOTAL OF NUMBER OF CHECKS WRITTEN
G	=	BALANCE OF BANK ACCOUNT
H	=	SGN(G)

By knowing what everything stands for, it is much easier to trace the steps of the program and figure out what is happening. Anytime that you are working with an unfamiliar BASIC program, the very first thing to do is to list and identify all the variables, then refer to this list as you track the program. This advice may not solve all your problems in understanding a program, but it will certainly help.

One other piece of advice: if you come across a variable that you can not identify, don't be alarmed. Some "computer people" quite often throw in unneeded variables to confuse someone who is trying to figure out their program. If you should encounter what appears to be a "UFV", unidentifiable fabricated variable, just take it out of the program and see if the program will RUN without it. If its absence is not noticed through a number of varied RUNs, you've probably located an "UFV". If that's the case - discard it.

Now back to our program, I don't use "UFVs" so don't waste time looking for them. Let's look at some of the specific programming techniques in this program.

1. The YES/NO method for determining the direction of the program.

```
40 INPUT "HAVE YOU MADE A DEPOSIT ", A$
50 IF  A$  = "NO"  THEN  130
```

If your answers here had been NO the computer would have skipped the entire section relating to deposits. If your answer is YES the program then proceeds to the next LINE. This same method is used again starting with LINE 180.

```
180 INPUT "HAVE YOU WRITTEN ANOTHER CHECK ? ", Y$
190 IF Y$ = "YES"  THEN  140
```

Here the program can either proceed to the next step or go back for more information. If the answer is YES, the program goes back to LINE 140 for the required INPUT. If the answer is NO, then the computer has all the necessary information to finish.

Some programmers prefer to use just the abbreviation for YES and NO --- "Y" and "N". If you use this method you must adjust your If statement accordingly.

2. The "value accumulation" technique.

LINE 80 and LINE 160 both use this technique to get a total of another variable. Only in the world of computer can something be equal to itself plus something else. Let's look at what's happening:

```
70  INPUT "WHAT IS THE AMOUNT OF THE DEPOSIT  :", E
80  F = F + E
```

Everytime the computer gets a new value for the variable E it then goes to LINE 80 and that value is then added to the previous value of the variable F. If after LOADing our above program into the computer and before telling the computer to RUN, we typed in PRINT F, we would get 0. So:

START: F = F + E then F = 0

If we then responded to the INPUT statement by typing in 10 for the value of E, then:

F = F + E then F = 10

because: F = 0 + 10

Then if we had another INPUT for E of 5:

F = F + E then F = 15

because: F = 10 + 5

Then if we had another INPUT for E of 12:

F = F + E then F = 27

because: F = 15 + 12

and so on, and so on....

3. "COUNT THE PASSES" technique:

This is essentially the same thing as above, except that instead of keeping a running total of a second variable as we did in that example, we are just counting the number of times that the program passed by a given LINE. If we know the reason for passing by "that" LINE, then we know how many "causes" there were which required that passage. In other words, if you knew that a certin man only passed by your office door to go to the water cooler, and that he passed by your door 7 times in the course of a day -- you know that he went to the water cooler seven times that day.

This same logic is applied in LINE 90 and LINE 150.

```
90  K = K + 1
150 C = C + 1
```

Each time the computer passes either one of these LINEs, the value of its respective variable will increase by 1. Variable K counts the number of deposits that are made. Each time a deposit is INPUT, the program passes through LINE 90. Each time the program passes through LINE 90, K is increased by 1. If you know how many times the computer passed through LINE 90, you know how many deposits were made.

These are all of the major programming techniques used in the above program, but there are some other items which might require some explanation. Consider LINE 200:

```
200  G = A + F - D
```

G = BALANCE OF CHECKING ACCOUNT

So, if you take your "starting balance," A, and add the "total amount of deposits," F, and subtract the "total amount of checks," D, you will end up with your "balance," G.

LINEs 260, 270, and 280, are what this whole chapter is all about. This is the SIGN function in action. Depending on the value of SGN(G) your friendly "banker" will either give you "thumbs up" or "thumbs down."

```
260  LET H = SGN(G)
270  IF H = 1  THEN  320
280  IF H = -1 THEN  350
```

What this tells the computer is that if the balance of you bank account is "positive" then GOTO LINE 320 and tell you. If the balance of your account is "negative" then GOTO LINE 350 and tell you. If you are exactly even, no money in the bank and not overdrawn, then proceed to LINE 290, and tell you that.

The computer will evaluate the value of the variable G, which in this case is the "balance of the checking account" as determined by LINE 200. Then depending on the SIGN of the value of G (+,-,0), a value is assigned to the variable H (1,-1,0). The value of the variable H really directs the activity.

One last footnote to this discussion of LINE 260, this LINE is called a LET statement. As you will notice I have not included the "LET" in the above program LINE 260, but I did include it in the explanation of LINE 260. In a LET statement the computer doesn't care if you put the word LET in or leave it out.

LINE 360 has another heretofore unexplained function in it. This is called the ABSOLUTE FUNCTION. The ABSOLUTE function does only one thing, it returns the "absolute value" of the variable it is instructed to "function" with. In this case G. In our program G will have a negative value, as it is found in LINE 360. That's how we got to LINE 360,

our account was overdrawn. Since the program is telling you how much to deposit to cover the overdrafts, it must be a "positive" or "absolute" value. The "absolute value" of a number is "that" number, with complete disregard to its sign.

LINEs 230,240,330,and 360 all have funny little sets of symbols,letters, and numbers enclosed by commas (%$C10F2). These are not typing errors, they are format instructions to the computer.

That's more than you probably wanted to know about SGN(A) ....

22

THE CHAIN COMMAND

The CHAIN command when used in a LINE of a BASIC program tells the computer to STOP the program that it has in memory, and LOAD another program and start executing the "new" program. The procedures for doing this are no different than if you were to do it yourself, by putting the computer in the READY condition and then type in LOAD "PROGRAM", and then type RUN. The only difference is, that by using the CHAIN command you have switched from "manual" to "automatic." It's always nicer for the computer to do the work than to have to do it yourself.

Consider these two silly examples:

The first program we will name BIRDS

```
10  FOR  V9 = 10 TO 19
20  ! " BIRDS " ,
30  NEXT V9
40  CHAIN "BEES"
```

The second program we will name BEES

```
10  FOR  Q5 = 1 TO 12
20  ! " BEES ",
30  NEXT Q5
40  CHAIN "BIRDS"
```

Assuming that both of these programs are on the same mini-disk, unless you have a multi-disk system, you need only LOAD either BIRDS or BEES and tell the computer to RUN. The computer will keep the BIRDS and BEES coming for as long as you would like. In fact, the only way to STOP them would be to abort the cycle, by a CONTROL-C or turning the computer off. The CONTROL-C is the preferred method.

If we RUN we get:

```
BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS
BEES BEES BEES BEES BEES BEES BEES BEES BEES BEES BEES BEES
BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS BIRDS
BEES BEES BEES BEES BEES BEES BEES BEES BEES BEES BEES BEES
```

and so on, and so on, and so on ....

What is happening is the the computer is LOADing "BIRDS" and executing that program, then the CHAIN command in LINE 40 tells the computer to LOAD "BEES" and execute that program, then the CHAIN command in LINE 40 of that program tells the computer to LOAD "BIRDS" and RUN that program. This cycle is set up by the CHAIN commands in both programs and will continue until you STOP it.

There are obviouly some very practical uses for the CHAIN command. If you had a business program which prepared your invoices, this invoicing information could then automatically be passed on to the "accounts receivable" program, and then this information could automatically be passed on to the "general ledger" program, and from that to the "income statement" program, and then to the "balance sheet" program. All of this could automatically be done by CHAINing the necessary programs in the sequence that you wanted.

If you have a need for the CHAIN command, the possibilities for its use are further extended when used in conjunction with other STATEMENTs such as the ON statement or the IF statement.

So that's what the CHAIN command does ....

23

THE DATA STATEMENT

Data does not have to be stored on a mini-disk or any other type of "device" to be available for use by the computer. It can be part of the program as DATA STATEMENTS. When DATA is supplied by this method, it is given a LINE NUMBER, just as is any other STATEMENT in a program. The DATA STATEMENTs can be placed anywhere in the program, but are usually placed at the end of the program.

The same general rules which we have already discussed for using READ statements with the mini-disk, apply to READ statements used with the DATA statement.

When the computer is told to READ a value for a variable, it goes through the DATA STATEMENTs in numerical/sequential order and "picks up" a value. This process of READing DATA is almost exactly like the process when READing data from a mini-disk or other "device." Much like the pointer on the disk reader starts at the "start of FILE" and sequentially READs the data, advancing itself at the rate that data is READ, until the pointer reaches the "end of file" or END MARK; so does the computer's "DATA statement pointer."

When the computer is told to READ data from a DATA STATEMENT, it READs the DATA starting with the DATA STATEMENT with the lowest LINE NUMBER and sequentially READs all the DATA on that LINE, from left to right. For example:

```
10 READ F
15 IF F = 0  THEN 60
20 PRINT F,
30 GOTO 10
40 DATA 1,2,3
50 DATA 4,5,6,0,7,8
60 ! "END OF DATA"
70 END
```

If we RUN we get:

```
1 2 3 4 5 6 END OF DATA
```

Each value for the NUMERIC VARIABLE F was READ by the computer,

and then that value was PRINTed. The computer started READing DATA from LINE 40, the first value for F was 1, the second was 2, and the third time a value for F was READ it was 3. Then when there was no more DATA on LINE 40, the computer proceeded to the next DATA STATEMENT --- LINE 50. It then READ each value from that LINE.

Note that each value (data) is separated by a comma and that the last item of DATA on each LINE is not followed by a comma.

If we were to change our READ statement to:

```
10  READ F,B
```

and leave everything else as it is in the above program, we would get:

```
1 3 5 END OF DATA
```

The first time LINE 10 is excuted:

```
                        F = 1    B = 2
the second time:        F = 3    B = 4
the third time:         F = 5    B = 6
the fourth time:        F = 0    B = 7
```

If we were to change the READ statement to:

```
10  READ  F,B,H
```

If we RUN we get:

```
1 4 END OF DATA
```

The first time LINE 10 is excuted:

```
                    F = 1     B = 2     H = 3
the second time:    F = 4     B = 5     H = 6
the third time:     F = 0     B = 7     H = 8
```

Now let's consider our "FRUIT" program that is used to explain SEQUENTIAL FILEs. If you will remember, we wrote a program to prepare a DATA FILE and also wrote a program to READ that DATA FILE and when the product number was INPUT the computer would PRINT the name of the fruit. For a program that short, and with no more DATA than was used, it would have been more reasonable to use the DATA STATEMENT method. This is how the program would be:

```
 10  INPUT "WHAT IS THE PRODUCT NUMBER : ",P
 20  IF P = 0 THEN  160
 30  READ N,F$
 40  IF N = P  THEN  70
 50  IF N = 0  THEN  100
 60  GOTO 30
 70  PRINT  P,"    ",F$
 80  RESTORE
 90  GOTO 10
100  ! "THERE IS NO PRODUCT WITH NUMBER : ", P
110  RESTORE
120  GOTO 10
130  DATA 100,"APPLES",200,"ORANGES",300,"LEMONS" ,400
140  DATA "BANANAS",500,"PEACHES",600,"GRAPES",700
150  DATA "PLUMS",800,"CHERRIES",0,"ZERO"
160  END
```

If we RUN we get:

WHAT IS THE PRODUCT NUMBER : if we type in: 600

we get: 600 GRAPES

WHAT IS THE PRODUCT NUMBER : if we type in: 200

we get: 200 ORANGES

WHAT IS THE PRODUCT NUMBER : if we type in: 0

we get: READY

Two things should be noted about the above program. The first is that the READ statement must exactly match the TYPE of DATA available to be READ. It must be an exact match --- number for number and "word" for "word"...

The second item worthy of further consideration is the RESTORE STATEMENT found in LINEs 80 and 100. The RESTORE statement would be comparable to the OPEN statement when using a data FILE. This statement RESTORES the "pointer" to the start of the DATA (to the "start of FILE").

Each time you enter a "product number" for the INPUT statement, the program starts its search of DATA from the start of the DATA statements. This is because we have told the computer to RESTORE all DATA before it goes back to LINE 10, the INPUT statement. Had we not included the RESTORE statement, then our search for all successive INPUT "product numbers" would start in the middle or end of our DATA, wherever the "pointer" happened to be after its last READ statement. This would generate false OUTPUT (information that the computer PRINTs on the monitor) similar to that experienced in our example "Account Number/Active/Non-active," when discussing OPENing and CLOSing data FILEs.

Also, if you were to review the section on SEQUENTIAL FILEs, you will note that our programs on "AUTOPARTS" is very similar to the above "FRUIT" program, and it too could easily be done with DATA STATEMENTs. In fact, in my opinion, it's a lot easier and faster to work with DATA STATEMENTS rather than DATA FILEs. This even becomes more evident when one has to make a change to some of the already generated DATA. With the DATA STATEMENT all you have to do to make a change is EDIT the LINE. With a DATA FILE you must RANDOM ACCESS that specific DATA you want to change and write a program to change it.

If the DATA STATEMENT method is quicker,faster, and easier to use, why then would anybody hassle with DATA FILEs? On the surface that is a very valid question, but if one considers how much "space" is taken up just storing DATA, the answer becomes very obvious. All of the cited examples contained only a limited number of DATA items. If our "auto parts house" wanted to inventory 6000 items instead of just the 10 we used in our example, we would run out of available memory before we ever got started. The same would be true for any of the other examples. These are all very small and short programs, almost useless in a "real" situation.

However, that does not render the DATA STATEMENT method useless. There is both a place and a need for this technique, and your time spent in perfecting your use of it will be well rewarded. It has been for me.

That's all there is to DATA STATEMENTS......

24

THE NUMERICAL FORMAT

Anytime you have numerical OUTPUT, whether it is "hard copy" (computer talk for printed material) or it is displayed on the monitor, it is formatted. That is to say, that the numbers are "printed" in a definite method of display. If you don't format numerical OUTPUT, the computer automatically will. If the computer does it, all numerical OUTPUT is printed in the DEFAULT FORMAT.

There is nothing fancy about the DEFAULT format. It is very functional and very simple. It is also the most commonly used format for numbers. In the DEFAULT format all displayed numbers contain up to eight digits. This maximum total of eight digits is used with complete disregard as to where the decimal point is placed. If the number to be displayed is greater than eight digits, only the first eight numbers are displayed. Any remaining numbers are rounded off to the eighth number or the number is printed in SCIENTIFIC NOTATION if it is too large or too small to round off within the eight digit limit and still maintain most of its value.

Here are some examples of numbers supplied to the computer and then printed in the DEFAULT formatd:

INPUT	DEFAULT FORMAT
1000	1000
350200509	3.5020051E+08
123.12345678	123.12346
.123456789	.12345679
.0000000005	5E-10
12345678919	1.2345679E+10

There are instances when the DEFAULT format will work, but a different format would make the data easier to use.

Consider this program:

```
10 INPUT " NUMBER OF BARRELS OF OIL ON HAND : ", J
20 INPUT " CURRENT COST PER BARREL          : ", B
30 INPUT " CURRENT SHIPPING COST PER BARREL : ", A
40 ! " VALUE OF OIL ON HAND             : ", J*B
50 ! " TOTAL COST OF SHIPPING           : ", J*A
60 ! " TOTAL NUMBER OF GALLONS ON HAND : ", J*55
70 END
```

If we RUN we get:

```
NUMBER OF BARRELS OF OIL ON HAND : we type in: 1273
CURRENT COST PER BARREL          : we type in: 18.395
CURRENT SHIPPING COST PER BARREL : we type in: 1.859

VALUE OF OIL ON HAND            : 23416.835
TOTAL COST OF SHIPPING          : 2366.507
TOTAL NUMBER OF GALLONS ON HAND : 70015
```

All of the answers are correct, but it would be easier to understand and more useful if it were better formatted. It would be much better if it had dollar signs in the proper places, right justified, commas, and decimal points for cents. All of this can be accomplished by a few simple changes.

The necessary changes are all made in the PRINT statements.

Let's first discuss LINE 40:

```
40 ! " VALUE OF OIL ON HAND  : ", J*B
```

If we change LINE 40 to:

```
40 ! " VALUE OF OIL ON HAND  : ", $$C10F2, J*B
```

If RUN with the same INPUT as above, we would get:

```
VALUE OF OIL ON HAND  :  $  23,416.84
```

As opposed to the previous OUTPUT:

VALUE OF OIL ON HAND : 23416.835

Now let's look at what took place. The key to the new format is the addition of "%$C10F2" to LINE 40. This addition gave us a dollar sign, comma in the proper place, two places after the decimal rounded off, right justified, and space for up to ten digits. The " %$C10F2 " is called a FORMAT STRING.

The % . . percent sign in a PRINT statement tells the computer that format instructions are forthcomming.

The $ . . dollar sign tells the computer to PRINT a dollar sign in front of the data to be PRINTed.

The C . . letter C in the FORMAT STRING tells the computer to PRINT commas in the proper places for the numeric values.

The 10 . . number ten in the FORMAT STRING tells the computer that there are a total of 10 characters in the OUTPUT.

The F . . letter F in the FORMAT STRING tells the computer to print the numbers in FLOATING POINT format.

The 2 . . number two tells the computer how many places beyond the decimal you want PRINTed and how you want the numbers rounded off.

The STRING FORMAT must be enclosed in commas in the PRINT statement. Any features that you do not want can be omitted from the STRING FORMAT. If you did not want the comma to appear in the OUTPUT, but you wanted everything else to remain the same, your STRING FORMAT would be:

%$10F2

If you wanted the comma, but did not want the dollar sign:

%C10F2

If you did not want either the dollar sign or the comma:

%10F2

If you wanted the dollar sign, the comma, and three places beyond the decimal point:

%$C10F3

If you wanted everything exactly as above but you need space for 15 total characters instead of the formatted 10:

%$C15F3

If you only wanted the 15 character range to be right justified and there would be no numbers beyond the decimal:

%15F0

As you can see, you can take or leave as much of the STRING FORMAT as you want. You can expand its scope to handle very large numbers with dollar signs and commas or diminish it to only right justify whole numbers or decimals.

In our program above we would change the other PRINT statements to:

```
50 ! "TOTAL COST OF SHIPPING          : ",%$C10F2,J*A
60 ! "TOTAL NUMBER OF GALLONS ON HAND : ",%C10F0,J*55
```

In this program we have included a STRING FORMAT in every PRINT statement. Many programs may have hundreds of PRINT statements all requiring the exact same STRING FORMAT. It would work all right if you included the same STRING FORMAT in each and every PRINT statement, but it is not necessary if you make one addition to the STRING FORMAT. If we were to add the "number" sign (#) to the STRING FORMAT:

%$C#10F2

All subsequent numbers would be formated by the PRINT statement which contained the above STRING FORMAT. To recap:

```
%$C10F2    FORMATS A SINGLE PRINT STATEMENT
%$C#10F2   FORMATS ALL FOLLOWING PRINT STATEMENT
```

There may be some OUTPUT in a program that you don't want formatted like all the rest. Consider our program above, we would not want our "number of gallons" to be formatted like dollars and cents. On a short program as above it was easier to add the STRING FORMAT to each PRINT statement, but suppose you had a program that had one hundred PRINT statements, half of which were formatted one way, the other half used the DEFAULT format. All you have to do is to "void" the STRING FORMAT by the following:

```
65 ! %#
```

The %# symbols in combination may be added to any PRINT statement or may be included on a LINE by themselves as in LINE 65 in the above example, either way the effect is the same -- they cancel all previous STRING FORMATs.

Now that we are back where we started, we have covered the subject .....

25

THE EXIT STATEMENT

Experience has shown me that the EXIT statement is seldom included when one is originally writing a program which contains a "nested LOOP," but that it is always added as a result of a CONTROL STACK ERROR. It is one of those things that nobody ever thinks about until they need it. The important thing is not to completely forget it, thus spending needless hours trying to figure out why a program won't RUN.

Consider the CONTROL STACK ERROR as a reminder. The absence of an EXIT statement is not the only thing that will generate this ERROR, but it's one ot the things that will.

The EXIT statement allows one to EXIT a "nested LOOP" (a LOOP within another LOOP) before the computer has completed its cycle of that LOOP, and then be able to re-enter that same LOOP as if it did complete its cycle.

This, of course, can be best explained by example. So here it is . . . . .

Consider this program:

```
10 FOR X = 1 TO 3
20 PRINT "X = ",X
30 NEXT X
```

If we RUN we get:

```
X = 1
X = 2
X = 3
READY
```

This is exactly as expected. If we change the program to the following:

```
10 FOR X = 1 TO 3
20 ! "X = ",X
25 IF X = 2 THEN 40
30 NEXT X
35 END
40 ! " I AM OUT "
50 GOTO 30
```

If we RUN we get:

```
X = 1
X = 2
  I AM OUT
X = 3
READY
```

As you can see, we started the LOOP, then left it, then re-entered it right where we left off. So far no ERROR and no EXIT statement. If we change our program to:

```
 5 FOR Y = 1 TO 3
10 FOR X = 1 TO 3
20 ! "X = ",X
25 IF X = 2 THEN 40
30 NEXT X
35 NEXT Y
38 END
40 ! " I AM OUT "
50 GOTO 35
```

If we RUN we get:

```
X = 1
X = 2
  I AM OUT
CONTROL STACK ERROR IN LINE 35
READY
```

Well, we kept messing around until we finally got an ERROR. All we tried to do was to RUN the same program three times that we had already sucessfully RUN one time.

The other thing that we tried to do was to leave a "nested LOOP" before it had finished its cycle, and then tried to re-enter it. When you do this you will always get a CONTROL STACK ERROR.

Since the computer is very obedient, and does exactly as it is told, our program has put it in the position of having to execute two conflicting directives at the same time. One, it is told to finish its X LOOP for X = 1 TO 3. The other, it is told to get the NEXT Y. If it goes to the NEXT Y it can't finish its X LOOP. If it finishes its X LOOP it can't go to the NEXT Y. So, the computer says to hell with the whole thing --- giving you a CONTROL STACK ERROR.

All that we have to do to make everybody happy is to tell the computer that when necessary, it is all right if it EXITs the X LOOP and does not finish its cycle. This allows the computer to finish its X LOOP when it can, and leave it when it can't.

To do all of this, we add the EXIT command to LINE 25.

```
25 IF X = 2 THEN  EXIT  40
```

If we RUN we get:

```
X = 1
X = 2
 I AM OUT
X = 1
X =2
 I AM OUT
X = 1
X = 2
 I AM OUT
READY
```

One practical use of this new gained knowledge would be to use it in what I call the "unique data test" programming technique. This programming technique allows one to take DATA from any source -- INPUT statements, READ/DATA statements, or from READ/DATA files -- and test each piece of data to see if it is the same as any previous data or is it unique. If its unique it is assigned a unique variable, if it is not, it is not.

Here is the program:

```
                              10  DIM A(4)
         ----------------  20  FOR T = 1 TO 4
        1                     30  READ A
data    1   unique ------  40  FOR D = 1 TO 4
feed    1     test 1          50  IF A = A(D) THEN EXIT 90
loop    1     loop  ------  60  NEXT D
        1                     70  K = K + 1
        1                     80  A(K) = A
         ----------------  90  NEXT T
     unique data  -----  100  FOR C = 1 TO K
        print     1          110  PRINT A(C)
        loop        -----  120  NEXT C
                             130  DATA 10,20,10,30
```

By changing the DIMENSION of A, this program can be expanded to handle as many values of A as available memory will allow. By suitably changing the source of "feed" for the variable A, this information could be supplied by a wide range of methods.

Now let's look at the program as written. The sequence of events for this program are:

1. From the first cycle of the "feed loop" the computer gets a value for A.

2. It then sends this data to the "test loop". If the data is equal to any previous value for the variable A, the computer goes back to the "feed loop" and gets another value for A.

3. If the value of A does not equal any previous value, it is then assigned a unique variable name -- A(K) = A. The computer is then sent back to the "feed loop" to get another value for A.

Because I feel that this is an important programming technique, since comparison of data is a key computer function, and because I spent two days trying to figure this whole thing out, and because I think that a complete understanding of this program has other things to offer, I shall fully explain how it works. Here it is as I see it.

If you RUN you get:

****

The FIRST TIME the computer cycles the program:

```
LINE 20                 T = 1
LINE 30                 A = 10



                                        D        A(D)
LINE 40  unique                         1        A(1) = 0
LINE 50   test       A = 10             2        A(2) = 0
LINE 60   loop                          3        A(3) = 0
                                        4        A(4) = 0
```

SINCE A IS NOT EQUAL TO A(D)
A = 10 < > any A(D)

The computer finishes its "unique test loop"
and goes on to the next LINE --- LINE 70.

```
LINE 70             K = 0 + 1     K = 1

LINE 80             A(K) = A      A(1) = 10


LINE 90        Go back to "feed loop" for another A.
```

****

The SECOND TIME the computer cycles the program:

```
LINE 20                 T = 2
LINE 30                 A = 20
```

```
                              D         A(D)
LINE 40                       1         A(1) = 10
LINE 50       A = 20          2         A(2) =  0
LINE 60                       3         A(3) =  0
                              4         A(4) =  0
```

```
              A  IS NOT EQUAL TO  A(D)
                 A = 20 < > and A(D)
```

The computer has completed the "unique test loop" and goes to the next line of the program -- LINE 70.

```
LINE 70         K = 1 + 1       K = 2

LINE 80         A(K) = A        A(2) = 20


LINE 90         Back to the "feed loop" for another A.
```

****

The THIRD TIME the computer cycles the program:

```
LINE 20                       T = 3
LINE 30                       A = 10
```

```
                              D         A(D)
LINE 40                       1         A(1) = 10
LINE 50       A = 10          2         A(2) = 20
LINE 60                       3         A(3) =  0
                              4         A(4) =  0
```

```
          SINCE  A  IS  EQUAL  A(D)
                    A = any A(D)
                   10 = A(1) = 10
```

The computer is sent back to the "feed loop" for another A.

****

The FOURTH TIME the computer cycles the program:

LINE 20	T = 4
LINE 30	A = 30

		D	A(D)
LINE 40		1	A(1) = 10
LINE 50	A = 30	2	A(2) = 20
LINE 60		3	A(3) = 0
		4	A(4) = 0

SINCE A IS NOT EQUAL TO A(D)
A = 30 < > any A(D)

The computer finishes the "unique test loop" and goes to the next line --- LINE 70.

LINE 70 K = 2 + 1 K = 2

LINE 80 A(K) = A(3) = 30

The "feed loop" is now complete (T = 4) so the computer progresses to the next line -- LINE 100.

But first let's look at:

	D	A(D)
K = 3	1	A(1) = 10
	2	A(2) = 20
	3	A(3) = 30
	4	A(4) = 0

```
LINE 100    the
LINE 110   print       FOR C = 1 TO 3      ( K = 3 )

LINE 120    loop
```

```
Therefore:              C       A(C)
                        1       A(1) = 10
                        2       A(2) = 20
                        3       A(3) = 30
```

So we get: 10 20 30

****

I am sure that this is the longest chapter in any published book on the EXIT statement. For that reason, one should feel that it has been adequately explained -- I hope so.........

26

NUMERICAL LISTS AND ARRAYS

When you see DIM A(12) or A(4,7) or A(16), you see the "call letters" or the DIMENSION for a NUMERICAL LIST or a NUMERICAL ARRAY. Whenever I see them I feel a great urge to reprimand the computer industry for not making the format of these different from DIM A$(12) or A$(4,7) or A$(16). If this book does nothing more for you than to engrave in your memory that the two have little or nothing in common, it will be worth the cost of the book in the savings in time and frustration.

I caution you not to make the mistakes that I originally made by assuming that because they look alike, they therefore share the same qualities. This would be a normal assumption, since many other NUMERICAL and STRING qualities are shared. To compound the erroneous assumptions, there is little or nothing to indicate otherwise. You start getting hints that somthing is awry when you start using them.

Rather than going into detail of how each NUMERICAL and each "like" STRING differ, I prefer to discuss how each works. This chapter is devoted to the NUMERICAL, the STRING will be discussed elsewhere.

Let's first discuss the NUMERICAL DIMENSION statement. It appears in the BASIC program as:

DIM A(16) or DIM A(15),B(12) or DIM A(5,8)

The default DIMENSION of any NUMERICAL list is ten (10). Thus if you have a list of ten or less numbers you do not have to include a DIMENSION statement in your program. If you want to limit a list to a definite number of entries then you must include a DIMENSION statement.

Consider these examples:

```
10 FOR X = 1 TO 6
20 INPUT Y(X)
30 NEXT X
40 FOR X = 1 TO 6
50 PRINT Y(X),
60 NEXT X
```

If we RUN we get:

```
        we type in:
?           10
?           20
?           30
?           40
?           50
?           60
10 20 30 40 50 60
READY
```

If we typed: PRINT Y(4)

and pressed the RETURN KEY

we would get: 40

Now if we took the very same program and added:

```
5 DIM Y(3)
```

If we RUN we would get an OUT OF BOUNDS error if we added more than three values for Y. On the other hand if we changed LINE 5 to:

```
5 DIM Y(15)
```

The program would RUN exactly as it did in our first example.

What DIM Y(5) really means is that you have reserved in the computer's memory room for five values of Y. If you do not use all the reserved spaces, the memory is still reserved. For that reason you should not excessively over-DIMENSION any variable. There will be times that you will not know how many values a DIMENSIONed variable will have. On those occasions I recommend a DIMENSION close to what you would expect.

Since the computer will store the first five values of Y (DIM Y(5)), we can not use Y as our variable name, because Y will actually have 5 values. Also it's important to know exactly which value of Y we are dealing with. There are several ways to do this, but the most used is the subscript.

Such that:

```
The first value of   Y = Y(1)
The second value of  Y = Y(2)
The third value of   Y = Y(3)
The fourth value of  Y = Y(4)
The fifth value of   Y = Y(5)

The  K th  value of  Y = Y(K)
```

Where K in this example can equal 1 to 5.

If you had two or more variables which had multiple values, you would then have two or more variables in your DIMENSION statement. For example, if you had four variables which would have multiple values, and you needed to keep up with each value for each variable, and you knew the expected number of values each variable would have, your DIMENSION statement would be:

```
10 DIM A(15),B(25),C(7),D(100)
```

Note that the DIM need only appear one time and that the variables and their DIMENSION are separated by commas. The only other hard and fast rules that I can think of for the DIMENSION statement are:

1. A variable must be DIMENSIONed before it is used.

2. A variable can only be DIMENSIONed one time in a program, and can never be reDIMENSIONed in the body of the program.

The next item is the NUMERICAL ARRAY. It is like the NUMERICAL LIST only different. The DIMENSION statement for a NUMERICAL ARRAY which is to contain three rows of values and four columns of values for a specified variable would be:

```
10 DIM A(3,4)
```

This means that A is expected to have a total of 12 values, and that the computer is to store each separate value of A in an ARRAY made up of three rows and four columns, as illustrated.

Consider this:

```
                         COLUMN
                  1        2        3        4
              +-------+-------+-------+-------+
     ROW 1    I  10   I  15   I  20   I  25   I
              +-------+-------+-------+-------+
     ROW 2    I  30   I  35   I  40   I  45   I
              +-------+-------+-------+-------+
     ROW 3    I  50   I  55   I  60   I  65   I
              +-------+-------+-------+-------+
```

This is our ARRAY filled with DATA. There are many ways to fill an array, from specific location INPUT:

```
A(3,2) = 55
```

from values generated from a program:

```
90 LET A(2,4) = K
```

In the above case K=45

Or we can write a program specifically designed to fill our array. That program would be:

```
 10 DIM A(3,4)
 20 FOR X = 1 TO 3
 30 FOR Y = 1 TO 4
 40 INPUT A(X,Y)
 50 NEXT Y
 60 NEXT X
 70 ! "THE ARRAY IS FILLED"\!
 80 ! "WHAT BOX DO YOU WANT ?"
 90 INPUT "WHICH ROW      : ",X
100 INPUT "WHICH COLUMN   : ",Y
110 ! "A(",X,",",Y,") = ",A(X,Y)\!
120 GOTO 80
```

This program will start with the first ROW and fill it with DATA and then the second, and the the third.

If we RUN we get:

?

(In the interest of space type in the following numbers in sequence for each ?.)

10 15 20 25 30 35 40 45 50 55 60 65

THE ARRAY IS FILLED

WHICH BOX DO YOU WANT ?
WHICH ROW   : we type in: 2
WHICH COLUMN : we type in: 3
A(2,3) = 40

This program will keep RUNning as long as you are willing to type in numbers. If you decide to STOP, press the CTRL key and the C key at the same time. This is called a CONTROL-C and will abort the program.

That's how LISTs and ARRAYs work, but that's not all there is to know about them. The hard part starts now, learning how to use them. I can show you what they are and what they do ..... the rest is up to you ......

27

THE SUBSTRING

Now we are starting to get into the heavy stuff. I wasn't ready to write this section until today. I spent about two hours with my "analyst" (computer type) to make sure it really was like I thought it was. He said I was close enough to the truth to proceed. So we will.......

One of the biggest problems with understanding SUBSTRINGS, parts of "words," is getting them confused with other things. The format for the SUBSTRING - A$(3,7) - looks similar to the format for a NUMERICAL ARRAY - A(3,7). Since they do look so much alike, and you must admit they do, I have a tendency to equate traits of one with the other. This leads to mass confusion, and erroneous conclusions. Thus the need for my time with the "analyst."

By now, you must be asking why would anybody care about parts of "words." Though you may not be conscious of it, a lot of what we do with words involves working with their parts. When you look up the spelling of a word in the dictionary, you approach it in parts - the first syllable, then the next, and the next, until you "match" the word. When we alphabetize a list of names, we compare the first letters, then the second, and so on. When the computer performs these same tasks, it has to be told exactly how to do it, although you may be doing it subconsciously.

Consider the following:

```
STRING VARIABLE :   A$ = "RED WINE"

                    RED WINE
                    12345678

SUBSTRING:      A$(5) =  WINE
```

A$(5) is the 5th through the last character

A$(J) is the Jth through the last character

```
If J = 5   then   A$(J) =  WINE
If J = 8   then   A$(J) =  E
If J = 1   then   A$(J) =  RED WINE
```

A$(3,7) is the 3rd through the 7th character

A$(3,7) = D WIN

For our example STRING: RED WINE

A$(5) = A$(5,8) = WINE

Since by definition the first value enclosed by parentheses is the starting character in the SUBSTRING, and the second number is the ending character, and the STRING is always counted from left to right, the first value must always be less than or equal to the second value.

A$(8,5) will generate an OUT OF BOUNDS error

A$(I,J) is Ith character through the Jth character.

Where I <= J   (I is less than or equal to J)

```
If   I = 2  and  J = 6   then  A$(I,J) = ED WI
If   I = 1  and  J = 3   then  A$(I,J) = RED
If   I = 5  and  J = 8   then  A$(I,J) = WINE
If   I = 7  and  J = 7   then  A$(I,J) = N
```

A$(K,K) will always equal the Kth character.

```
If  K = 2  then  A$(K,K)  =  E
If  K = 5  then  A$(K,K)  =  W
If  K = 1  then  A$(K,K)  =  R
```

A$(3,3) will always equal the 3rd character = D
A$(7,7) will always equal the 7th character = N

Consider this program:

```
10 T$ = "ABCDEFGH"
20 FOR C = 3 TO 5
30 PRINT T$(C)
40 NEXT C
```

If we RUN we get:

```
CDEFGH
DEFGH
EFGH
```

Consider this program:

```
10 B$ = "ABCDEFGH"
20 FOR  E = 1 TO 3
30 FOR  F = E + 2
40 PRINT B$(E,F)
50 NEXT E
```

If we RUN we get:

```
ABC
BCD
CDE
```

Consider this program:

```
 10 READ A$
 15 IF A$ = "0" THEN 110
 20 ! " I'M THINKING OF A WORD - WHAT IS IT "
 30 INPUT "TYPE IN A THREE LETTER WORD : ",B$
 40 FOR X = 1 TO 3
 50 IF B$(X,X)=A$(X,X) THEN !"CORRECT  : ",B$(X,X)
 60 NEXT X
 70 !
 80 IF B$=A$ THEN !"**** YOU ARE RIGHT ****"
 85 !
 90 IF A$ < >. B$ THEN 30
 95 GOTO 10
100 DATA "CAT","DOG","PUT","TAP","COY","MIX","WRY"
110 END
```

This program compares each character of the INPUT word (B$) with each character of the "game word" (A$). If the computer makes a match, then the character is printed. If the computer does not make a match, it goes on to the next character and repeats the process. After it has evaluated all three characters (letters), it then evaluates the word. If it makes a match, it tells you so, and then READs a new "game word." If it does not make a match, the computer returns to the INPUT statement for you to try another word.

The above explanation and sample programs will only give you a start on the use of SUBSTRINGS. Learning how to use them well will widen your available choices to compare data. A good deal of the use of the computer is to compare data, the more ways you know how to do it, the more use you will get out of the computer.

That is NOT all there is to know about that .....

28

SEQUENTIAL FILES

To get the maximum use of your system it is mandatory to learn how to use data FILEs (TYpe 3 FILEs), and be familiar with how they work and how the computer READs and WRITEs to them. Any time you use (access) a data FILE you are storing data on a device and therefore are leaving FREE available memory in your computer for writing or expanding your program. You are also able to store data far beyond the memory capacity of your system, since each mini-disk will store about 90,000 bytes (88 K). As mentioned earlier, each mini-disk contains 350 blocks and each block contains 256 bytes.

Before getting into the specifics of how data FILEs work, a general over-all view of their operation will show that they are not too different from any other TYpe FILE. TYpe 3 data FILEs, like all other TYpe FILEs, must first be CReated. In order to do that they must be named. The same rules for naming other TYpes of FILEs apply for naming TYpe 3 FILEs. The next step is to TYpe the FILE: all DATA FILEs are TYpe 3. That means that they can only be used (accessed) by a READ or WRITE statment in a BASIC program. After you have done all the above, the FILE is ready for putting in data.

To add data to a FILE one usually writes a program that asks for specific information to be typed in (INPUT). Once the required information is supplied, the computer is then told to WRITE this information (data) onto the mini-disk. Each new addition is added to the mini-disk immediatly following the previous data, thus a sequential FILE is slowly filled with data. The required information may be no more than a single numerical value or may be more than a thousand numbers or words, depending on the program written to add (WRITE) data to the FILE. Regardless of the INPUT the computer essentially handles it all in the same manner, and records (WRITEs) it on the mini-disk the same way.

Just having a mini-disk full of information is not of much value. It only becomes valuable when one can ask for a specific piece of data and the computer in its unique way can search all that available information and then give you what you ask for, if it has it, in a matter of seconds. In order to be able to do this, you must now write a second program - this time to take out information (READ).

Having learned how to do all the above, you have greatly expanded the capacity of your system and increased its range of usefulness. Now let's learn how to do all the above.

First we must again remember that the computer does no more than what one could do himself by conventional methods. It has a couple of advantages over doing the task by "hand." Number one, it can search a FILE much faster. It essentially does it by the same method that you would: it looks at all the data until it finds some that matches what it is looking for.

Number two, the computer can evaluate the "found" data for any number of instructed conditions, such as; IF this is found - do this, or IF this is equal to that - do this, and so on.

The computer can perform all of these tasks in a matter of seconds, usually with little or no error. The human can perform all of these tasks by "hand," hopefully with little or no error. The main difference between the two methods is time and efficiency.

We will approach the use of DATA FILEs on the basis of contrasting the "hand" method with the "electronic" method. This will allow the reader to better understand what is happening, and be able to relate to the sequence of events.

Let's suppose that you work for an auto parts supply company. You have been given the job of preparing an inventory listing of all auto parts and assigning a part number to each. This information is to be placed in a file and stored for future use.

You start by first getting a file folder and naming the folder AUTO PARTS. You then get some paper and start recording the part number with its accompanying auto part. You start at the top of the page and continue to add part number, then part name. You continue this procedure until you have finished the task. You then put the papers in the file folder and then put the file folder into a file cabinet.

Using our above example, the "electronic" method would be essentially the same. First, we must CReate a file folder. We do this by the following.

1. We must put the computer into the DOS mode.

    a. From start up - put a mini-disk with Disk Operating System (DOS) on it in the disk-reader.

        On > type in: EX E900

b. From BASIC mode:

On READY type in: BYE

2. To CReate our "file folder," with the name AUTOPART, we must place an INITIALIZEd mini-disk in the disk-reader and then:

   On * type in: CR AUTOPART 300

   This CReates a FILE with the name AUTOPART which is 300 blocks long - which is almost the capacity of a new mini-disk.

3. We must then tell the computer that the FILE named AUTOPART is a TYpe 3 data FILE, so that we will be able to READ and WRITE to the FILE. This is accomplished by:

   On * type in: TY AUTOPART 3

We now have a "file folder" labeled AUTOPART, it is ready to put our RECORDs into it.

Next we want to develop a data FILE which contains two pieces of information for every auto part maintained in the inventory of a parts house. We want to give every auto part a part number, and then we want to identify the auto part which goes with this number. We want to WRITE this information in an orderly manner, as was done by the "hand" method above.

Our first step is to write a program which will add this information to our FILE in an orderly manner. There are two questions that the computer must ask: 1. "WHAT IS THE PART NUMBER ", 2. "WHAT IS THE PART NAME".

In BASIC these questions are represented by the INPUT statement as:

```
50  INPUT  "WHAT IS THE PART NUMBER  : ", N
```

Where INPUT is the alert word - reserved word - used by the computer to tell it to ask for some information. That portion of the INPUT statement which is in quotation marks is for the user's benefit. It is the method used by the computer to tell the operator what information he is to type in (INPUT). Anything that is enclosed in quotation marks following an INPUT statement is PRINTed by the computer. If no quoted remark follows the INPUT statement then a question mark (?) will appear on the monitor, indicating to the operator that they must type in some information before the computer can go on.

The most significant character in the above INPUT statement to the computer is the letter "N". The operator will never see this, but it is to the computer what the quotation mark enclosed portion of the INPUT statement is to the operator. The letter "N" is used to represent a numerical value - a variable - to the computer. It could have been any letter of the alphabet or any combination of letter and number from 0 to 9, i.e., K4,B2,X7,G,S ....

Let's assume that the first part number is 1001, and it is the part number for HY-TEMP SPARK PLUGS. When the computer ask you:

WHAT IS THE PART NUMBER :

You respond by typing in the number: 1001

You have assigned the numeric value of 1001 to the numeric variable "N". If you were to ask the computer what is the value of "N" it would PRINT 1001. "N" will always have this value until you change it.

The next question that computer should ask is "WHAT IS THE PART NAME". This question is represented by the INPUT statement:

```
60  INPUT  "WHAT IS THE PART NAME  : ", N$
```

The same general information pertaining to LINE 50 goes for this LINE with one notable exception -- the dollar sign, "$", which follows the "N". In pure computer talk this INPUT statement wants you to type in a STRING. For some unknown reason all alphabetic or alphanumeric values are called a
STRING. For ease of understanding, I have

found it useful to automatically substitute WORD every time I see or hear STRING. This conversion is not exactly valid, but by the time you understand the reasons why it's not, you will understand the concept of STRING variables.

Now back to LINE 60. On the monitor we would see:

WHAT IS THE PART NAME :

You would then type in (INPUT) : HY-TEMP SPARK PLUGS

We have now assigned the "word" - HY-TEMP SPARK PLUGS - to the "word variable," N$. Or in computer talk the STRING VARIABLE, N$, is equal to: HY-TEMP SPARK PLUGS.

The relationship between the two variables:

N = 1001

N$ = HY-TEMP SPARK PLUGS

is essentially the same as far as the computer is concerned. They are handled the same way, as illustrated above for the numeric variable, if we told the computer to PRINT N$ it would PRINT: HY-TEMP SPARK PLUGS, since that is what N$ is equal to.

The computer now has the required information for performing the task that we set out to do, i.e., values for both variables, or answers to both its questions. We must now tell the computer what to do with this information (data).

We want it to WRITE this unique combination of part number and part name on a mini-disk so that "1001" and "HY-TEMP SPARK PLUGS" are inseparable. If one knows the part number, the computer can tell you the part name or if one knows the part name, the computer will tell you the part number.

If we were doing this by the "hand" method, we would have to go to the file cabinet (the mini-disk), open the cabinet and take out our file folder for AUTO PARTS (OPEN the FILE named AUTOPART), write down the information on a sheet of paper in the folder (WRITE N,N$), and then close the file and put it back into the file cabinet (CLOSE), and we are finished (END). The computer must be instructed to do essentially the same thing. We tell the computer to:

```
40 OPEN #0, "AUTOPART"
```

This STATEMENT tells the computer to:

a. Go to "file cabinet" #0.
   Turn on the disk-reader.

b. OPEN the "file cabinet."
   Search for a FILE.

c. Find the "file folder" named "AUTOPART".
   Finds and OPENs the FILE - AUTOPART.

At this point the computer has the required information to be recorded, has located the proper FILE to WRITE the information to, has OPENed the FILE and is READY to put it in the FILE. We must now tell the computer to put the data into the "file folder." So our next STATEMENT is:

```
80 WRITE #0, N,N$
```

The "#0 " makes sure that the computer WRITEs the values for N and N$ to the same FILE that it spent all that time locating and OPENing. So, that's what it does : it WRITEs the values of the variables in the FILE represented by #0. If we could see the FILE AUTOPART we would see:

```
1001 HY-TEMP SPARK PLUGS *
```

The computer will automatically WRITE the values of N, N$ at the start of the FILE. The asterisk at the end of the value of N$ is my representation of an END MARK. After each WRITE command the computer will place an END MARK at the end of the last item it recorded - it is like a period at the end of a sentence. The need for the END MARK will become evident as we go on.

We are now ready to add more data. The next item will have the part number 1002, and it is a SEALED BEAM LIGHT. Therefore N = 1002 and N$ = "SEALED BEAM LIGHT". In order to get back to the start of our program we have to tell the computer to go back and get some more data. Since we did not tell the computer to CLOSE the FILE, it left it OPEN and the recording head (pointer) is at the end of the previous INPUT.

This being the case, we will not have tell it to do all that again. All we have to tell the computer to do is go back for more data or INPUT. So, we tell it to go to LINE 50, which is the first of the two INPUT statements:

```
90 GOTO 50
```

The next thing to appear on the monitor is:

```
WHAT IS THE PART NUMBER :
```

You would then type in: 1002

```
WHAT IS THE PART NAME :
```

You would then type in: SEALED BEAM LIGHT

The computer would then go through the same motions as before and our FILE would now look like this:

```
1001,HY-TEMP SPARK PLUGS,1002,SEALED BEAM LIGHTS*
```

Then the computer would go back for more data:

```
INPUT

WRITE

GOTO INPUT
```

This cycle would continue as long as you want it to. After a few minutes our FILE would look like this:

```
1001 HY-TEMP SPARK PLUGS 1002 SEALED BEAM LIGHTS
1003 BONDED BRAKE LINING 1004 STEEL BELTED TIRES
1005 SEAT BELTS 1006 LOW PRESSURE OIL GAGE 1007
AIR HORN 1008 HI-LOAD SHOCK ABSORBERS 1009 LOCK
TYPE GAS CAP 1010 WIDE ANGLE MIRROR*
```

That's exactly what we want. But it's not all that easy. If we were to PRINT the contents of our FILE AUTOPART as we have constructed it, we would not get what we have shown above, but all of our STRING variables (words) would be cut short and limited to only 10 characters (10 bytes).

Thus our first item:

1001 HY-TEMP SPARK PLUGS

would be:

1001 HY-TEMP SP

There is a pre-set limit to the size (DIMENSION) of any STRING variable (word). Unless told otherwise the computer will give all STRING variables the DIMENSION of 10 characters. If the size, length, or DIMENSION of a "word" is greater than 10 characters, only the first 10 characters are accepted by the computer and the rest are ignored.

Although this 10 byte limit would not effect some of the data entries in our FILE, it would render others unintelligible. Unless one is willing to play word games every time he gets only a portion of a "word," this situation would be considered unacceptable.

There is a method of correcting the problem. One need only consider the length (DIMENSION) of the longest single "word," STRING variable, in his data. In our case that would be 63 gharacters found in part number 1007. We must then tell the computer how many characters to reserve for each of our "words" (STRING variables). This is called DIMENSIONing a variable. Our DIMENSION statement is part of the BASIC program and must appear befor the variable is used. Once a variable is DIMENSIONed within a program that variable will always have that DIMENSION. You can not change that DIMENSION in a later portion of the program.

Our statement would be:

```
20 DIM N$(23)
```

This tells the computer to always reserve 23 bytes any time it READs or WRITEs the STRING variable N$. No matter how long the "word" for N$ is the computer will always allot 23 characters for it. For STRINGS that are shorter than this the computer will still use 23 characters, the unused portion being filled with blank spaces.

Now, let's stop and look at the BASIC program that we have thus far put together:

```
20  DIM N$(23)
40  OPEN #0, "AUTOPART"
50  INPUT "WHAT IS THE PART NUMBER : ", N
60  INPUT "WHAT IS THE PART NAME   : ", N$
80  WRITE #0, N,N$
90  GOTO 50
```

That looks and works great, it does everything that we set out to accomplish. I want to turn off the computer and go home for the day. Before I can leave I must put all the FILEs back and CLOSE the FILE cabinet. The computer is no different, it's got to do the same thing -- and like lots of people it must be told. We tell the computer by this statement:

```
100 CLOSE #0
```

That's all there is to that. But wait, if we examine the program we see that the computer will never get to LINE 100. Everytime it gets to LINE 90 we have told it to GOTO LINE 50. There doesn't seem to be any escape. We could just turn the thing off, but that's not recommended procedure. If we just turned it off while it is in a program loop (cycle) it may not have completed all its assigned tasks, and we may not get recorded all the data entries we need. This is how we do it:

```
55  IF N = 0 THEN 100
```

What this tells the computer is if ever the value for the variable N is equal to zero then GOTO LINE 100. So when the computer asks:

```
WHAT IS THE PART NUMBER :
```

If you type in: 0

The computer will GOTO LINE 100 and then the computer will CLOSE the FILE as instructed. This is called an ESCAPE statement.

But, that's still not all, there's one more thing that we should tell the computer, to give it one last chance to do everything it was supposed to do.

Because the computer is "lazy," it will store up all its instructed jobs and then do them all at once. This ability to "put off doing now, what you can do later" is not really a sign of laziness. It allows the computer to limit disk activity, so that the disk-reader is not turning on and off every second or so. The computer has a built in BUFFER, which is a small memory "cell" that stores data that it is to WRITE. If you were to just turn off the computer as earlier mentioned, you would not give the computer an opportunity to clean out its BUFFER memory. All the information still stored there would be lost.

For the most part, all of the BUFFER activity would have been completed at LINE 100, the CLOSE statement. Although it may not be required, I suggest that the last STATEMENT for any program be:

```
110 END
```

This shuts everything down in proper fashion and also tells anyone who LISTs the program that that last LINE is in fact the last LINE.

29

HOW THE COMPUTER READS A TYPE 3 DATA FILE

I continue to dwell on the "ins and outs" of data FILEs, because their sucessful use is dependent on one's total understanding of how they actually work. This attitude is probably a reflection of the difficulity that I experienced when I was first learning how to use them. On many occasions I was unable to get a FILE READ without generating an ERROR. In the absence of understanding what was going on, it was almost impossible to correct the ERROR.

Let's assume that we have CReated a FILE with the name "FRUIT". If we were able to see the contents of our FILE on the mini-disk it would look like this:

```
100 APPLES 200 ORANGES 300 LEMONS 400 BANANAS
500 PEACHES 600 GRAPES 700 PLUMS 800 CHERRIES*
```

The program used to fill this FILE with data could contain an assortment of INPUT and WRITE statements. The most obvious would be:

```
10  INPUT "WHAT IS THE FRUIT NUMBER : ", F
20  INPUT "WHAT IS THE FRUIT NAME   : ", F$
```

and then

```
90  WRITE #0, F,F$
```

If these LINEs were part of a program similar to the program that was used to fill the data FILE "AUTOPART" earlier, they would generate "FRUIT" FILE as shown above. But, the same FILE could be generated from multiple INPUT and WRITE statements.

Consider the following:

```
 30 INPUT "WHAT IS TWO TIMES 50 :", J
 40 INPUT "WHAT IS NEW YORK'S FAVORITE FRUIT :", P$
 60 INPUT "HOW MANY YEARS IN A BICENTENNIAL : ", T
 80 INPUT "WHAT IS FLORIDA'S FAVORITE FRUIT : ", Q$
 90 D = J + T
100 INPUT "WHAT FRUIT GOES BEST WITH SEAFOOD : ", Y$
160 WRITE #0, J,P$,T,Q$,D,Y$
```

Assuming the expected answers for the INPUT, if we were to look at the data on the FILE we would see:

```
100 APPLES 200 ORANGES 300 LEMONS*
```

Even though our INPUT and WRITE statements were radically different from the previous example, you'll note that the net result is identical. Both FILEs contain the same data in the same sequence. The computer is only assigning values to variables and then recording that information.

The same thing holds true for READing a data FILE. Using again our example FILE "FRUIT", all of the READ statements below when included in a properly written program will yield the same results.

```
  5  OPEN #0, "FRUIT"
 10  READ #0 , C
 20  PRINT C
 30  READ #0 , X$
 40  PRINT X$
 50  READ #0, E
 60  PRINT E
 70  READ #0, G$
 80  PRINT G$
 90  READ #0, M
100  PRINT M
110  READ #0, S$
120  PRINT S$
130  CLOSE #0
140  END
```

If we were to RUN a program which contained these LINEs and used the FILE "FRUIT" as its data source, we would get:

```
100
APPLES
200
ORANGES
300
LEMONS
```

Or using the same logic, if our program contained:

```
 10 OPEN #0, "FRUIT"
 50 READ #0, C,C$,W
 60 PRINT C,C$,W
 80 READ #0, H$,Y,D$
 90 PRINT H$,D$
 95 PRINT Y
100 CLOSE #0
110 END
```

We would get:

```
100 APPLES 200
ORANGES LEMONS
300
```

Or for a final example, again using the same premise:

```
10  OPEN #0, "FRUIT"
40  READ #0, A,A$,B,B$,C,C$
50  PRINT C$,B$,A$
60  PRINT A,B,C
70  CLOSE #0
80  END
```

We would get:

```
LEMONS ORANGES APPLES
100 200 300
```

The point is that READing data FILEs is not limited to the same INPUT that put data into that FILE. Nor must the format of OUTPUT follow the same sequence. The only "hard and fast" rule that must be followed is that data must be READ in the exact same order that it is contained in the FILE.

This last statement is so important that I feel an obligation to repeat it.

ALL DATA IN A DATA FILE MUST BE READ IN EXACTLY THE SAME ORDER THAT IT IS CONTAINED IN THE FILE.

If were were to examine our example FILE "FRUIT", we would notice that the sequence of order of data is:

```
NUMERIC VALUE   STRING VALUE   NUMERIC VALUE
STRING VALUE   NUMERIC VALUE  STRING VALUE
```

and so on .....

Therefore, we must READ our FILE in that same order. If our first READ statement in a program for this FILE was:

```
30 READ #0, R$
```

We would get:

```
TYPE ERROR
```

Which would indicate that we ask the computer to READ a value for the STRING variable R$ and the first data on the FILE is a NUMERIC VALUE --- thus the wrong type of data --- thus the TYPE ERROR.

In order to sucessfully READ a data FILE you must know the sequence of TYPEs of values contained in that FILE. Your READ statement must exactly match the available data to be READ. If it does not match, you will get a TYPE ERROR.

Consider the following program:

```
 10  INPUT " WHAT IS YOUR FILE NAME : ", F$
 20  OPEN #0, F$
 30  IF TYP(0) = 0  THEN  120
 40  IF TYP(0) = 1  THEN  60
 50  IF TYP(0) = 2  THEN  90
 60  READ #0, S$
 70  PRINT " STRING ",
 80  GOTO 30
 90  READ #0, N
100  PRINT " NUMBER ",
110  GOTO 30
120  PRINT " END OF FILE "
130  CLOSE #0
140  END
```

This program will READ any FILE and PRINT the sequence of value TYPEs. Thus if we were to use it to READ our example FILE "FRUIT", we would get:

```
NUMBER  STRING  NUMBER  STRING  NUMBER  STRING
NUMBER  STRING  NUMBER  STRING  NUMBER  STRING
NUMBER  STRING  NUMBER  STRING  END OF FILE
```

This program is able to READ any FILE without knowing the sequence of TYPEs of data by the fact that we have made provisions for what the computer is to do no matter what TYPE of data it encounters. This is done with the TYP FUNCTION.

So much emphasis has been placed on the sequence of TYPE of data in the data FILE because that is how it is READ by the computer when accessed SEQUENTIALly. Remember how the "pointer" always starts at the beginning of the FILE when it is OPENed, and advances through the FILE at the rate of each READ statement.

The pointer will not go back to the start of the FILE unless you reOPEN the FILE. If you fail to remember this fact and you are writing a "search" type program where you want to repeatedly search a FILE for different kinds of information, you will end up only searching that portion of the file that has not yet been READ. The computer will only READ the data that is left after the last advance of the "pointer" (reader arm). Therefore after each "search" you must CLOSE the file and the OPEN it again before starting the next "search." Let's consider the following example.

Let's say that we have CReated a FILE named "ACCOUNTS".

From BASIC:

```
on  READY  or  CURSOR  type in: BYE
on  *   type in: CR ACCOUNTS 10
on  *   type in: TY ACCOUNTS 3
```

Next we write a program to add DATA to the FILE named ACCOUNTS. ACCOUNTS will contain all the active accounts of an imaginary company. Here is the necessary program:

```
10 OPEN #0, "ACCOUNTS"
20 INPUT "ACTIVE ACCOUNT NUMBER : ",S
30 IF S = 0 THEN 60
40 WRITE #0, S
50 GOTO 20
60 CLOSE #0
70 END
```

Then we RUN the program and add all the account numbers for all of the active accounts. Now, our file ACCOUNTS looks like this:

```
100 200 300 400 500 600 700 800 900 *
```

Now let's write a program to put all of the above to some practical use. The purpose of our program is to be able to type in an account number, and the computer will tell us if that is an active account.

```
 10  INPUT " WHAT IS THE ACCOUNT NUMBER : ", X
 20  IF X = 0 THEN  140
 30  OPEN #0, "ACCOUNTS"
 40  IF TYP(0) = 0 THEN  110
 50  READ #0 , P
 60  IF P = X  THEN 80
 70  GOTO 40
 80  ! "ACCOUNT NUMBER ",X," IS ACTIVE ."
 90  CLOSE #0
100  GOTO 10
110  ! "ACCOUNT NUMBER ",X," IS NOT ACTIVE ."
120  CLOSE #0
130  GOTO 10
140  CLOSE #0
150  END
```

If we RUN we get:

```
WHAT IS THE ACCOUNT NUMBER : we type in: 400
ACCOUNT NUMBER 400 IS ACTIVE.

WHAT IS THE ACCOUNT NUMBER : we type in: 260
ACCOUNT NUMBER 260 IS NOT ACTIVE.

WHAT IS THE ACCOUNT NUMBER : we type in: 0
READY
```

If we were to take out LINE 90 and then RUN we would get:

```
WHAT IS THE ACCOUNT NUMBER : we type in: 400
ACCOUNT NUMBER 400 IS ACTIVE.

WHAT IS THE ACCOUNT NUMBER : we type in: 200
ACCOUNT NUMBER 200 IS NOT ACTIVE.
```

Since we know what is on the FILE we know that 200 is an ACTIVE account. What went wrong? Account 200 would also like to know. They are sure they paid their bill.

By taking out LINE 90 we did not CLOSE the FILE, therefore the "pointer" (reader arm) will start READing the FILE from position "400". From that "point on" it will not encounter the number 200, but will reach the END MARK " * ", thus indicating that there is no active account, "200".

If we were to put LINE 90 back in our program and take out LINE 120 all would go well until we had an account that was not "active." Then, all future INPUT would result in a TYPE ERROR. The reason being that the "pointer" would be at the END OF FILE and the next thing to come up would be the END MARK --- * , which is neither a NUMERICAL VALUE or a STRING VALUE. The other possibility would be that the computer would find no "active" accounts.

(I have taken certin liberties in the above explanations in the interest of best explaining the effects and need of certain program LINEs. The fact is, that by taking out the above mentioned LINEs the programs probably would not RUN, since we would be telling the computer to OPEN a FILE that is already OPEN. Also, the format of all the PRINT statements would not be actually PRINTed as I have illustrated them, but the content of these statements would be as represented. I do not

feel the need to be totally exact and dilute the reader's concentration by mentioning all the possible exceptions. These "half truths" will become evident as you better understand the principles of the subject.)

That completes this section on "How the computer READs a SEQUENTIAL FILE." Study this section and remember what it says. Without knowing the information presented above, you will experience many frustrations as a result of not being able to get the computer to READ a FILE without generating a plethora of FILE ERRORs, as I did......

30

HOW TO ACCESS AND USE SEQUENTIAL DATA FILES

Now that we have CReated a data FILE and we have put data into it, it's time for it to become a useful tool and pay its way. Up to this point everything that we have done with regards to SEQUENTIAL FILEs has been for the computer. We've got nothing out of it. We must instruct the computer as to how it can give us the information we want, when we want it. This, of course, is done by writing another BASIC program.

When we start to write this program we must remember that the computer is generally dumb, it can't do even the simplest thing unless we tell it to. The nice part about the computer is that we need only tell it once, and it doesn't forget.

First, let's consider what we want the computer to do with the data FILE that we have CReated.

a. Find a data FILE named AUTOPART

b. OPEN it

c. Search for some specific information

d. Retrieve the information if it's there

e. PRINT it out or display it on the monitor

f. Be READY to do it all over again

That's all we want the computer to do with our data FILE.

Following the same line of thought as we did before when we CReated our data FILE, we must first locate and OPEN the FILE.

We do that by:

```
20  OPEN #0, "AUTOPART"
```

Since we know that some of the "words" (STRING variables) in our data exceeds the DEFAULT DIMENSION of 10 characters (10 bytes), we must DIMENSION our STRING variable " N$ ".

So we write:

```
10 DIM N$(23)
```

We must decide what information we want and what information we must type in (INPUT). We can either give the computer the "part name" and ask it to find the "part number" that goes with that name, or vice versa. Let's say that we want to be able to type in (INPUT) the "part number" and the computer will supply us with the name of the part with that number. To do this we must supply the "part number." Our INPUT statement would then be:

```
30 INPUT "WHAT IS THE PART NUMBER : ", P
```

Now that the computer has the "part number," or in reality a value for the numeric variable P, that is:

P = Whatever "part number" that we type in

It is ready to go looking for a "part name" with a "part number" that matches the one we typed in. So we tell the computer to:

```
40  READ #0 , N,N$
```

Where:

N = PART NUMBER in our data FILE

N$ = PART NAME in our data FILE

Remembering what our data FILE looked like, the first time the computer READs the FILE it will come up with:

N = 1001

N$ = HY-TEMP SPARK PLUGS

We want the computer to compare the INPUT "part number," P with the stored data "part number," N. If the two numbers are the same, then we want the computer to PRINT the "part number," N and its accompaning "part name," N$. As complicated as it sounds it's not - here's how:

```
50 IF N = P THEN  PRINT  N,N$
```

This LINE tells the computer to do just what we said:

If N (FILE part number) = P (INPUT part number)
Then PRINT N (FILE part number) and N$ (FILE part name)

But what happens if the numbers don't match ? Since we haven't told the computer what to do in this situation, nothing will happen. The computer will just sit there, the disk-reader will run for a while, and then all will stop. Our next LINE will therefore be:

```
60  IF TYP(0) = 2  THEN  40
```

Since the "part numbers" didn't match the computer will immediately go on to the next LINE. LINE 80 tells the computer that if the next thing to come up on the mini-disk is a numeric value then GOTO LINE 60. The way it tells the computer all of this is from the READ TYP function:

```
TYP(0)
```

(I shall go into greater detail in explaining the TYP function in another chapter. Suffice it to say for the time being that it does what I have indicated that it does.)

So what happens is that if N is not equal to P the computer goes back to LINE 60 and READs another set (RECORD) of variables. It continues to do this until it finds a set of variables (N and N$) where N = P, when it finds N = P it then PRINTs the set (RECORD).

```
             INPUT  P

            READ  N, N$

N <> P                      N = P

            PRINT  N, N$
```

The next obvious question should be, "What happens if it doesn't find a match ?" Suppose that the INPUT number was never put into the data FILE, or the operator hit the wrong key and put in a "bad" number --- what then?

The first thing I can tell you is that if you didn't tell the computer what to do under these circumstances, it sure as hell doesn't know. Again, everything will come to a stop, or worse, the disk-reader will just keep running and running and running. I would suggest a CONTROL - C if this occurs.

You, as the programmer, must anticipate all the possible occurences, and write the program so that when they occur the computer knows what to do. To digress for a minute, I would like to relate a story I read in the newspaper which will illustrate my point.

> In a state which allows personalized license plates for automobiles an enterprising computer programmer requested the license plate "NONE." The agency which censors personalized plates saw nothing wrong with it, so the license plate "NONE" was issued.
>
> It did not become evident to the authorities why such a license plate should not have been issued, until about 200 unpaid parking tickets later.
>
> It seems that whenever an unpaid parking ticket was turned over to the warrant division, they used their computer to match the name and address of the owner of the car with the license plates.
>
> The computer was programmed such that if the license plate number was not a valid match, it would automatically give it the default number "NONE", thus making a match impossible.

Now, back to more important things. What to do if the computer can't make a match with our "part numbers?"

Do this:

```
70  IF TYP(0) = 0  THEN  80
```

This tells the computer that if the next thing up on the mini-disk is an END MARK, which would indicate END OF FILE, then the computer has gone through the entire FILE and has reached the END and still can not make a match. In that case the computer is instructed to GOTO LINE 150.

Which says:

```
80  PRINT " THERE IS NO PART WITH THAT NUMBER "
```

Then we start all over again, with:

```
90 CLOSE #0
```

To insure that we start our next search at the "start of file," as opposed to starting where the "file pointer" is now located, at "end of file."

And then:

```
100 GOTO 20
```

To start the whole process all over again.

Let's review our program up to this point:

```
 10  DIM N$(23)
 20  OPEN #0, "AUTOPART"
 30  INPUT "WHAT IS THE PART NUMBER : ", P
 35  IF P = 0 THEN 110
 40  READ #0, N,N$
 50  IF N = P  THEN  PRINT N,N$
 55  IF N = P THEN 90
 60  IF TYP(0) = 2 THEN 40
 70  IF TYP(0) = 0 THEN 80
 80  !"THERE IS NO PART WITH THAT NUMBER"
 90  CLOSE #0
100  GOTO 20
110  CLOSE #0
120  END
```

That is essentially the whole program. We have only two thing to add: 1. An escape statement. 2. CLOSE and END statements. Our "escape" statement would be similar to the one we used in the preceding chapter:

```
35  IF P = 0  THEN  110
```

Such that when the computer asks you:

WHAT IS THE PART NUMBER :

and you type in: 0

The computer will GOTO LINE 110, which says:

110 CLOSE #0

and then: 120 END

That's all there is to it.........

31

A CLOSER LOOK AT THE TYP FUNCTION

We have covered a very complex, but one of the most useful sets of tools the computer has to offer --- the use of SEQUENTIAL FILES. I need to go back and mention a few things which I think would be more salient now than at the time they first appeared in the program.

My explanation of the TYP function was inadaquate, but I did not want to divert from the point at hand. Therefore, I just briefly mentioned what the TYP function did in that particular program, without further explanation. Let's now look at the TYP function closer.

If you remember I said that any time you OPEN a FILE the "reader" (pointer) always goes to the "front" or start of the FILE. It then moves in an orderly manner through the FILE, much like the arm on a record player follows the groves in a phonograph record. There are many occasions when one does not want to start at the "front" of the FILE. Often it's necessary to start at the end of the FILE.

Let's consider the program we wrote to add data to our FILE AUTOPART. We have long since CLOSEd that FILE and put it away. Now a new shipment of auto parts has just arrived, and we must add these new items to the FILE. If we were to use the program we had previously written, we would have a minor disaster.

Why would we have a "disaster" when it worked so well before? The reason is, that by OPENing the FILE again the pointer (reader) starts at the beginning of the FILE, as it always does. If you were to RUN the program as written, without making any changes, you would be erasing previous data at the same rate that you were adding new data. You would be "over writing" anything that was already on the mini-disk and when you finished adding your new data, that's all that you would have --- your new data.

Even if you added only one item and the FILE already had 236 other data entries, you would still end up with a FILE with only one item on it. The reason is that when the computer finished writing that single item on the mini-disk, it would then place an END MARK after it. This of course, would tell the computer when it goes to READ the FILE, after it READs that single item and then encounters an END MARK, that that is the "end of file."

Now, all of this is all right if that's what you intended to do, but if you meant to add the data for the new auto parts at the end of all previous data entries, then you've got to tell the computer to do it that way. Here's how .....

Whenever the computer READs a data FILE on a mini-disk it always "looks" one step ahead --- it always knows what's coming up next. We are provided with a method to take advantage of this fact, it is called the TYP FUNCTION. Like all other FUNCTIONs discussed, it is part of the BASIC LANGUAGE program, and is there to be called upon when needed.

Let us look at our original program that we wrote to add data to our FILE AUTOPART:

```
10  DIM N$(23)
15  OPEN #0, "AUTOPART"
30  INPUT " WHAT IS THE PART NUMBER : ", N
40  IF N = 0  THEN  80
50  INPUT " WHAT IS THE PART NAME   : ", N$
60  WRITE #0, N,N$
70  GOTO 30
80  CLOSE #0
90  END
```

If we make the appropriate changes, such that we can add new data to the end of the old data, thus expanding our FILE AUTOPART. Our new program to add additional data to the FILE would look like this:

```
         -------  22  IF TYP(0) = 0  THEN  30
        I         24  IF TYP(0) = 2  THEN  26
        I         26  READ #0, N1, N1$
         -------  28  GOTO 22
                  10  DIM N$(23)
                  15  OPEN #0, "AUTOPART"
                  30  INPUT " WHAT IS THE PART NUMBER : ", N
                  40  IF N = 0  THEN  80
                  50  INPUT " WHAT IS THE PART NAME : ", N$
                  60  WRITE #0, N, N$
                  70  GOTO 30
                  80  CLOSE #0
                  90  END
```

As you can see, both programs are exactly the same except for the inclusion of LINEs 22,24,26,and 28. Let's examine exactly what each of these LINEs does and how they affect the program.

```
22  IF TYP(0) = 0 THEN  30
```

This tells the computer that IF the next thing to come up on the FILE is an END MARK, THEN go to LINE 30. LINE 30 is really the start of our program for adding data to our FILE after it is properly OPENed and DIMENSIONed.

```
24  IF TYP(0) = 2  THEN  26
```

This tells the computer that IF the next thing to come up on our FILE is a NUMERIC VALUE (number), then go to LINE 26.

```
26  READ #0, N1, N1$
```

The computer already knows that the next thing to come up is a NUMERIC VALUE, that's why it came to LINE 26. We already know that our FILE is constructed of sets (RECORDs) of a NUMERIC VALUE (part number), followed by a STRING VALUE (part name). Since we know how the FILE is constructed, we also know that the computer can not READ another NUMERIC VALUE until after it has READ a "part name," a STRING VALUE. Knowing all of this, we are in a position to tell the computer what to READ so that we know that the "next thing to come up" on the mini-disk will either be a NUMBER or an END MARK.

The other thing of interest which you may have noticed is that we have changed the "names" of both of the variables. N has become N1 and N$ has become N1$. We must differentiate between typed in (INPUT) values --- N, N$ and READ values --- N1, N1$. If we did not, the computer would already have a value for the "part number" and "part name," it would be the last set of values it READ from the mini-disk.

For this particular program it would not make any difference, since we immediately assign new values for N and N$ by our INPUT statements. It is generally not in your best

interest to use the same variable "names" for "procedural tasks," such as setting the "file pointer" to the proper position, as you use for "actual values" in your program. You may inadvertently assign a value to something without knowing it, particularly when you are working with long complicated programs.

28 GOTO 22

The only way the computer could have gotten to LINE 28 was to have not encountered an END MARK when it READ the FILE. Since we are trying to set the pointer (reader arm) to the "end of FILE" so that we can add additional new data, we know that we have not reached the "end of FILE" and therefore must send the computer back to READ more data. That is what LINE 28 does, it starts the READ cycle (loop) all over again. Thus we get:

CHECK FOR END MARK

NO	YES
CHECK FOR NUMBER	START WRITING
READ SET OF DATA	READ SET OF DATA

This cycle (loop) will continue until the computer finally encounters an END MARK. When the computer has READ the entire FILE and has placed the pointer (reader arm) on the eventually found END MARK, the file pointer is then in the proper position to WRITE new data at the end of all previous data. Thus starts the INPUT portion of your program and then the WRITE portion.

We can now add new data to this FILE at anytime without fear of "over WRITEing" any previous data. No more problems - right? Not so.... everytime we resolve one problem, we potentially create another one. Since the computer can not think for itself and must rely on you to do its thinking, it is at a disadvantage. We humans usually resolve immediate problems with little regard to long term effects, as evidenced by the solution of the above problem. What we did works and works well,for our particular set of circumstances. However, there is still one "flaw" in the system, or maybe I just haven't learned how to get around it.

If you were to take our "new improved" program above and start with a new unused mini-disk; our FILE properly CReated and TYped, everything exactly as it should be; and try to RUN it --- it wouldn't work.

The disk-reader would come on and just run and run and run ...... after a while the computer may turn it off or it may not. To put a STOP to this, press the CTRL key and the "C" key at the same time (CONTROL C). Everything will come to a STOP and READY will again appear on the monitor.

Now, let's see if we can figure out what has happened. Our approach to the solution must be guided by the fact that the computer is very unyielding and that it always does what it is told to do. Since we feel sure that we have not erred, as evidenced by the fact that we have already successfully RUN this program before, it must be a malfunction of the equipmeant --- our natural instinct is to blame that which we least understand. Not so .....

Granted, the computer does exactly as it is told to do, but if we fail to tell it to do something then it is left in the position of not knowing what to do. That is what we have done with our new, improved program. LINEs 62 and 63 tells the computer what to do if it finds a number or an END MARK. But nowhere have we told the computer what to do if it doesn't find anything. On a "brand new" FILE that has never been written to there would not be a number or an END MARK. Neither condition of our program would be met and we have not told the computer what to do under these circumstances.

Not only have we not told the computer what to do in a situation like this, neither has the manufacturer, nor has the "person" who wrote the Disk Operating System, nor the "person" who wrote the BASIC language program --- nobody told the computer what to do. So, don't feel like it's just our dumb mistake.

I would suggest another TYP FUNCTION to handle a situation like this, which would tell the computer that if it finds a "clean" FILE then go back to the "start of FILE" and continue with the rest of the program --- but, none presently exists. The problem however, does exist.

There are a couple of ways to program around the problem, both are essentially the same. One is that you can at the start of any program which is used to add data to a FILE, include an INPUT statement which ask: ARE YOU STARTING A NEW FILE ? If you type in: NO, then the program proceeds as normal. If you type in: YES, then the program is routed around the TYP FUNCTION statements and immediately goes to the INPUT and WRITE statements for the first data entry.

Such as the following:

```
17 INPUT " ARE YOU STARTING A NEW FILE ? ", Y$
18 IF Y$ = "YES" THEN 30
```

This method is quite effective, but requires that you respond to the INPUT question everytime you RUN the program. Since in my situation I don't start a new FILE that often, I prefer to just type in a GOTO statement on those occasions when I do start a new FILE. Thus I include:

```
18 GOTO 30
```

for just that first data entry. The program is written so that it never goes back to LINE 18 for all subsequent data entries. Since I don't SAVE the program with LINE 18 in it, the next time I LOAD the program it is not there.

There is a hazard in using this method, in that if you route your program around the TYP FUNCTIONs in any subsequent data entries, you will end up with only one data entry --- the last one you typed in. To insure that this doesn't happen, it's a good idea to STOP the program after the first data entry and DELete LINE 18. Then, the program will be exactly as originally written.

32

HOW TO COPY A DATA FILE FROM ONE MINI-DISK TO ANOTHER

There is no real problem with transfering a TYpe 3 data FILE from one mini-disk to another, it just has to be done by some method other than the LOAD and SAVE method used for programs written in BASIC, i.e., TYpe 2 FILEs.

Essentially, what you must do is ReaD a specific number of blocks of data at a specific starting place (ADDRESS) on the mini-disk into the computer, then put in the "new" mini-disk and WRite the same data to a specific Disk Address.

This method can be used to completely copy anything that is on one mini-disk to another, not limited only to TYpe 3 FILEs. You alternately ReaD a specific number of blocks into the computer's memory (number of blocks dependent on available memory) from the "old" mini-disk, and then WRite them out of the computer's memory at a specific Disk Address on the "new" mini-disk.

Here is how to do it for a TYpe 3 FILE:

1. Put the computer in the DOS mode.

   a. From start up type in: EX E900
   b. From BASIC: on READY type in: BYE

2. Put the mini-disk with the TYpe 3 FILE to be transfered in the disk-reader.

3. On * type in: LI

   Let's suppose that the FILE that we intend to copy is named CUSTOMER and it is 50 blocks long. When we LIst we get:

   ```
   CUSTOMER   4   50   3
   ```

   Which indicates that the FILE name is CUSTOMER, it starts at block 4 (disk-address), it is 50 blocks long, and it is a TYpe 3 FILE.

4. Now put the "new" INITIALIZEd mini-disk into the disk-reader and CReate a FILE named CUSTOMER that is 50 blocks long.

   To do this you:

   On * type in: CR CUSTOMER 50

5. We must tell the computer that the FILE with the name CUSTOMER is a TYpe 3 data FILE. You then:

   On * type in: TY CUSTOMER 3

6. So that we know where to tell the computer to WRite the stored data it got from ReaDing the "old" mini-disk, we need to know the Disk Address of the FILE named CUSTOMER on the "new" mini-disk To find that out we:

   On * type in: LI

We get:

CUSTOMER 135 50 3

7. Let's stop and recap what we've done and what we want to do. First look at the Disk Address:

"old"	CUSTOMER	4	50	3
"new"	CUSTOMER	135	50	3

   We want the computer to ReaD 50 blocks of data from the "old" FILE named CUSTOMER starting at Disk Address 4 and store it in its "regular" memory (READ,WRITE,LOAD,SAVE, and program portion of the memory). So that's what we tell it:

On * type in: RD 4 2A00 50

This command tells the computer to go to Disk Address 4 and ReaD the next 50 blocks and store this data in a "box" in the computer with the ADDRESS 2A00 (hex RAM address).

8. We now want the computer to WRite the 50 blocks of data that it has in its "regular" memory to a FILE on the "new" mini-disk (put it in the disk- reader) with the name CUSTOMER, whose Disk Address is 135. So that's what we tell the computer to do:

   On * type in: WR 135 2A00 50

   This tells the computer to go to a "box" that is located at ADDRESS 2A00 somewhere inside it. Get the contents, 50 blocks of data, from the "box" and WRite it on the mini-disk starting at Disk Address 135.

9. The best way that you can be sure that all went well is to RUN a program that accesses this FILE and uses the data.

   Another method is the use of the following program. I have named the program FILELIST and would recommend copying the program in BASIC, and SAVEing it as one of your general use programs.

```
10 REM   THIS PROGRAM PRINTS THE CONTENTS
20 REM   OF A TYPE 3 DATA FILE
30 DIM N$(300)
40 INPUT "TYPE IN FILE NAME  :  ", F$
50 OPEN #0, F$
60 IF TYP(0)=0 THEN ! "END OF FILE"
65 IF TYP(0)=0 THEN 140
70 IF TYP(0)=2 THEN 110
80 READ #0, N$
90 ! TAB (15),N$
100 GOTO 60
110 READ #0, N
120 ! TAB(6),N
130 GOTO 60
140 END
```

As I mentioned earlier this method of ReaDing and WRiting from one mini-disk to another is not limited only to transfering TYpe 3 data FILEs, but may be used to copy entire mini-disk including DOS, BASIC, and any other TYpe programs and their LIsting. However, it is the most practical method for TYpe 3 FILEs.

It is most important to develop a full understanding of this technique and be able to have it available when needed. As with other TYpes of programs, one should never have only one copy of an important program, this includes data FILEs.

33

HOW TO RANDOM ACCESS DATA FILES

That is not a question, but the title of this chapter. Not too many days ago it was a question for me. I had already decided not to include the subject, because it was outside the scope of this book. (That's literary talk for, " I don't understand it well enough to write about it.") I gave it one more try and all of the sudden, for the first time, it made sense.

I can now tell you that RANDOM ACCESS of data files is just as logical and just as easy as all the other phases of operating one of these computers, if you finally get to the point that you understand the principle of the thing. I had already read that RANDOM ACCESS was an advanced technique. This statement alone had made me decide not to spend the time to figure it out till I absolutly had to.

I finally had a reason. I have a very long data file which lists all the customers and their addresses for the company I work for. The file was set up so that the most frequent customers were at the front of the file and the occasional customers were near the end of the file. Since this file was READ by SEQUENTIAL ACCESS, it saved time by setting the file up this way. No sooner than I had entered all the names and addresses (about 200), than one of our very good customers moved. I was faced with the decision of either learning RANDOM ACCESS or reINPUT all of the data into a new file.

After reading the "computer manual" for the tenth time I almost optioned for the latter. Here is what the "computer manual" tried to say but never quite could ......

In general all you do is READ or WRITE to a specific place (ADDRESS) on the "mini-disk." To determine the starting point (ADDRESS) where you want the computer to READ or WRITE, you must by some method tell the computer how many "bytes" to skip over before it starts. Let's consider the simple READ statement:

```
10 READ  A,B,C
```

This READ statement tells the computer to READ a set of three numerical variables -- A,B, and C. In a DATA statement or in a DATA file this set of three numerical variables would exactly match the variables in our READ statement. This SET of numerical values is called a RECORD.

If you were able to see the data on a "mini-disc" it would appear as a series of sets (RECORDS) of three numeric values each.

Let's CReate a file, write a program, and WRite data to the file:

First CREATE a file named "FILE"

1. If you are in BASIC mode: ON READY type in: BYE
2. Turn on "disk-reader" insert INITALIZED "mini-disc"
3. On * type in: CR FILE 5
4. On * type in: TY FILE 3

This program will add DATA to our "FILE":

```
10 OPEN #0 , "FILE"
20 INPUT " VARIABLE  A  =  : ", A
30 IF A = 0  THEN  80
40 INPUT " VARIABLE  B  =  : ", B
50 INPUT " VARIABLE  C  =  : ", C
60 WRITE #0 , A,B,C
70 GOTO 20
80 CLOSE #0
90 END
```

If we RUN we get:

```
VARIABLE A = : we type in: 1
VARIABLE B = : we type in: 1
VARIABLE C = : we type in: 1
VARIABLE A = : we type in: 2
VARIABLE B = : we type in: 2
VARIABLE C = : we type in: 2
VARIABLE A = : we type in: 3
VARIABLE B = : we type in: 3
VARIABLE C = : we type in: 3
VARIABLE A = : we type in: 4
VARIABLE B = : we type in: 4
VARIABLE C = : we type in: 4
VARIABLE A = : we type in: 5
VARIABLE B = : we type in: 5
VARIABLE C = : we type in: 5
VARIABLE A = : we type in: 0
READY
```

Now our "FILE" is complete with DATA...

This "FILE" can be ACCESSED by either SEQUENTIAL or RANDOM ACCESS methods, as can most files.

Let's say that the data in the "FILE" looks like this:

```
RECORD #1   1    1    1
RECORD #2   2    2    2
RECORD #3   3    3    3
RECORD #4   4    4    4
RECORD #5   5    5    5
            I---------I  ("end mark")
```

Each RECORD in our "FILE" occupies a specific number of bytes.

REMEMBER: EACH NUMERICAL VALUE IS FIVE (5) BYTES

This is true regardless of the size of the numerical value i.e...number. 1 = 5 bytes and 1,000,000 = 5 bytes. Since our "FILE" has three numerical values in each RECORD, each RECORD contains: 3 times 5 bytes for a total of 15 bytes.

3 NUMERICAL VALUES X 5 BYTES EACH = 15 BYTES

The RANDOM ACCESS READ statement for our "FILE" would be:

```
10 READ #0  %15*X, A,B,C
```

The 10 . . .is the LINE NUMBER.

The READ #0 . .is the READ "command" to the computer to tell it that it is reading a "mini-disc" file.

The % . . . .is the "command" to the computer to evaluate the following arithmetic expression and "skip" that many bytes before it READs the specified values.

The   15 . . . .is the number of bytes in each set (RECORD).

The   X . . . .is the number of RECORDS (sets) to be skipped
It is also equal to one less than the RECORD NUMBER.

The A,B,C . . .is the names of three numerical variables that the computer is to READ values for.

Let's look at what we have:

X	RECORD #	A	B	C	ADDRESS	BYTES	TOTAL BYTES
0	RECORD #1	1	1	1	%15*0	15	15
1	RECORD #2	2	2	2	%15*1	15	30
2	RECORD #3	3	3	3	%15*2	15	45
3	RECORD #4	4	4	4	%15*3	15	60
4	RECORD #5	5	5	5	%15*4	15	75

I---------I

Now consider our READ STATEMENT again:

```
10 READ #0  %15*X ,A,B,C
```

This is the same as:

SKIP	15 BYTES	X	TIMES	THEN	READ	A,B,C
%	15	*	X	,		A,B,C

(I understand that the times sign (*) and the X get mixed up, but just read through it.)

Now let's consider some examples and use the above chart to follow our examples like the computer.

```
10 READ #0  %15*X , A,B,C
```

If we want to know what A,B, & C is in the third record we set X = 2. The computer will skip the first two RECORDS and then READ values into our variables from the third RECORD.

```
A  =  3
B  =  3
C  =  3
```

If we want to know what A,B, & C is in the first record we set X = 0. This tells the computer not to skip any RECORDS, but to READ the values into our variables from the very first RECORD.

```
A  =  1
B  =  1
C  =  1
```

If we want to know what A,B, & C is in the Jth RECORD we set the value of X = J - 1. Using the variable expression J - 1 to replace X, since they are the same, enables us to then type in the exact RECORD number we want to READ or WRITE. If our READ statement was:

```
10  READ #0  %15*(J-1) , A,B,C
```

If we set J = 3 (We want to READ RECORD #3) we get:

```
A = 3
B = 3
C = 3
```

A program that would RANDOM ACCESS this "FILE" would be:

```
10 OPEN #0 , "FILE"
20 INPUT "WHAT RECORD NUMBER DO YOU WANT : " J
30 IF J = 0  THEN 70
40 READ #0  %15*(J-1) , A,B,C
50 PRINT A,B,C
60 GOTO 20
70 CLOSE #0
80 END
```

If we RUN we get:

WHAT RECORD NUMBER DO YOU WANT : If we type in: 4

We get:

```
4
4
4
```

WHAT RECORD NUMBER DO YOU WANT : If we type in: 2

We get:

```
2
2
2
```

WHAT RECORD NUMBER DO YOU WANT : If we type in: 0

We get:

READY

To change a value in a RECORD (set) in our example "FILE" is a little more complicated, but not much. Let's first look at our "FILE" structure again:

```
RECORD # 1    1   1   1
RECORD # 2    2   2   2
RECORD # 3    3   3   3
RECORD # 4    4   4   4
RECORD # 5    5   5   5
              I---------I
```

Since we have already established that:

```
10 READ #0 $15(J-1), A,B,C
```

will allow access to a specific RECORD by setting the variable J equal to that RECORD number, let's review how we arrived at the access expression: $15*(J-1). The 15 was the sum of the "byte space" occupied by the three numerical values for A,B, and C.

```
A  =  5  bytes
B  =  5  bytes
C  =  5  bytes
```

If we set J = 3 for our READ statement the computer will send the "pointer" to the start of RECORD number 3 and READ the next 15 bytes for the values of A,B, and C. It would actually go to the start of RECORD number three and READ the first 5 bytes for the value of A, then READ the second 5 bytes for the value of B, and then READ the third 5 bytes for the value of C.

The fact that we know that that's how the computer will READ the "FILE" enables us to derive an ADDRESS for each individual value for A,B, and C.

The ADDRESS for A is : $15(3-1)
The ADDRESS for B is : $15(3-1)+5
The ADDRESS for C is : $15(3-1)+10

Remember, that all the ADDRESS does is to tell the computer where to start READing. So if we tell the computer to go to the ADDRESS for A it will automaticaly READ the next "five byte space" for the value of A. For this reason the ADDRESS for the variable A is the same as the ADDRESS for the RECORD.

The ADDRESS, "starting point", for variable B would be the same as for variable A plus the "byte space" used for variable A i.e.. five bytes. If the computer was told to READ variable B, using the ADDRESS given above, it would "skip" two fifteen byte RECORDS (3-1) and then "skip" five more bytes (+ 5) and then READ the next "five byte space" for the value of variable B .

The ADDRESS for variable C would be the same as for variable A plus the "five byte space" used for variable A plus the "five byte space" used for variable B, so the ADDRESS, "starting point", for variable C would be:

$15*(3-1)+10.

We now know how to get to each individual value for each individual variable. By changing our READ statement we can now ACCESS any single value for any single variable. If we wanted to know the value of B in the Jth RECORD of our "FILE" our READ statement would be:

```
10 READ #0 $15(J-1)+5 ,B
```

For variable C it would be:

```
10 READ #0 %15(J-1)+10 , C
```

For variable A it would be our original READ statement, except we will tell the computer to READ only variable A:

```
10 READ #0  %15(J-1) , A
```

Once you have learned how to get to a specific piece of data, changing it should present no particular problem. Just getting there was most of the battle. First we must write a program which will allow us to OPEN the "FILE", INPUT our change, ADDRESS the "FILE", WRITE our change, and then CLOSE the "FILE". The necessary program would be:

```
 10 INPUT "DO YOU WANT TO CHANGE  A , B , OR  C : ", W$
 20 INPUT "WHAT DO YOU WANT TO CHANGE IT TO      : ", T
 30 IF W$ = "A" THEN  P = 0
 40 IF W$ = "B" THEN  P = 5
 50 IF W$ = "C" THEN  P = 10
 60 INPUT "WHAT  RECORD #  DO YOU WANT TO CHANGE  : ",J
 70 OPEN #0 , "FILE"
 80 WRITE #0  %15*(J-1)+P , T , NOENDMARK
 90 CLOSE #0
100 END
```

Now let's track the program and see what's happening.

LINE 10 .. This INPUT tells the computer which variable you are going to change by assigning the variable P to the variable you type in - A, B, or C. The variable P is used in the ADDRESS to set the pointer "up" the proper number of additional bytes for the individual variables A, B, or C.

LINE 20 .. This INPUT assigns a value to the variable T which assumes the position of either A, B, or C, the one you wanted to change in LINE 10.

```
     30
LINE 40 .. These three LINEs assign a value to the variable P.
     50

LINE 60 .. This INPUT tells the computer how many RECORDS to
           "skip."

LINE 70 .. This statement OPENs a file named  "FILE".

LINE 80 .. This is the statement that essentially does all
           the work.  It takes the information from all the
           INPUT, determines the ADDRESS, then WRITEs the
           new value at that ADDRESS.
```

But what's that at the end of LINE 80 -- ? "NOENDMARK" ! If you will remember we previously said that after the computer executes a WRITE statement it automatically prints an "END-MARK". This indicates that this entry is the last entry of a WRITE statement.

We certinly don't want an "END-MARK" in the middle of a data file. It would do two things: Number 1, it would play havoc with our "byte count system" we have established to determine the ADDRESS, since the "END-MARK" would throw in an additional byte. Number 2, it would indicate to the computer if one were using the SEQUENTIAL ACCESS method that it had reached "end of file," when in fact it was in the middle of a "FILE".

If we did not tell the computer not to put an "END-MARK" after it finished its WRITE command, it automatically would. Since we don't want it we must include the "reserved word" NOENDMARK at the end of the WRITE statement.

Keep in mind, however, what would happen if we had to change the very last variable of the very last RECORD in the "FILE" ----- we would end up with NO END MARK at all. If we were to access this "FILE" by the SEQUENTIAL ACCESS method and you got to "end of file" and there was NOENDMARK the "disk-reader" would just keep running and running and running.

$This would not be a problem for the "FILE" that was only RANDOM ACCESSED since you would always tell the computer the specific address that you want it to go to.

This is not the last word on RANDOM ACCESS, but it will give you a working knowledge of how to use it for numerical variables.

Isn't there more to life than just numerical variables? There must be! Yes, Virginia, there is; there's STRINGS. STRING variables (words) present a greater challenge to RANDOM ACCESS of data files, but a challenge that you are very close to meeting.

With numerical variables the computer reserved a specific number of bytes for each variable. For the number three the computer reserved the same number of bytes as it would for the number three million. It may not make any sense but you can always "count" on it ---- and we did.

Since a string (word) variable can potentially range in size from one byte to 89,600 bytes, (maximum determined by total number of bytes that can be stored on a single "mini-disk," plus or minus 2 or 3) and the computer handles strings differently from numbers, it would be impossible to pre-size (dimension) a string variable.

The key word in the above "paragraph sentence" is the word DIMENSION ---- remember the DIM statement. To quickly review, an unDIMENSIONed string (word) has the default DIMENSION of 10 or, if you like, 10 bytes. If the string is shorter than 10 bytes, and you know it's an unDIMENSIONed string, this creates no particular problem. If the unDIMENSIONed string is longer than 10 bytes (characters) the computer will use only the first 10 bytes and ignore the remaining characters. Thus:

GO TO HELLISPORT

If left unDIMENSIONed becomes:

GO TO HELL

Which is not what we want. Remember each string variable must be DIMENSIONed or it will assume the default DIMENSION of 10 characters (bytes).

To determine the size of a file RECORD (set of variables) which contains one or more string variables, all we do is count the number of bytes (characters) in each string and then add 2 bytes for each string variable. Let's first CReate a file and write a program to WRite data to it:

1. Get in the disk operating system (DOS).
   A. From start up, load from "mini-disk": EX E900
   B. From BASIC mode, type in: BYE

2. ON * type in: CR FILE$ 5

3. ON * type in: TY FILE$ 3

4. Put the computer in BASIC mode.
   A. From start up: ON * type in: GO BASIC
   B. If BASIC has already been LOADed once
      ON * type in: JP 2A04

(In all instances where you "type in" always follow that by pressing the RETURN key - I forget to mention that sometimes)

The nice thing about numeric FILEs and RANDOM ACCESS is that there are very strict rules for the computer to follow that are part of the operating systems of the computer, the DISK OPERATING SYSTEM, and the BASIC LANGUAGE program. The programmer is not bothered by all the internal goings on and the how and why it works. If the programmer properly sets the disk in motion, the computer will do the rest. He need not be bothered with the "size" of any numeric value, how many bytes are required to store it, or adjusting the program in a specific order. All of these factors remain constant for all numeric values. If you know what to do with one numeric value, then you know what to do with all numeric values.

However, such is not the case with STRING variables and their relationship to TYPE 3 DATA FILES and RANDOM ACCESS. All of the sudden life becomes very complicated, and you have to pay attention to every minute detail. An error can produce irreparable consequences; resulting in erroneous values, if any, and could render a DATA FILE that may have taken months to prepare completely useless in just a few microseconds. How could such devastation be brought about by just one little error or miscalculation?

When you start writing to a DATA FILE by RANDOM ACCESS, at least two significant things happen: one, whatever you tell the computer to WRITE it writes, and two, wherever you tell the computer to WRITE it writes. There is no consideration on the computer's part as to whether the file pointer is at the start or end of file, or whether the sequence that you are WRITING is in the same format as other DATA in the FILE. When you tell the computer to WRITE something at some place in the DATA FILE by RANDOM ACCESS, the computer will obey and will overwrite anything at that place, even if it's right in the middle of another variable--it's done. This is why even a small mistake can be amplified a thousand times.

Now that I've made you almost afraid to mess with RANDOM ACCESSing of DATA FILEs which contain multiple STRING VARIABLES, I will tell you that if you fully understand what's going on you will never experience any of the above. In addition to that, I will also tell you that fully understanding what's going on is no big deal. Here's how to do it....

First you will need to remember a lot of the things that we have already discussed and used. Having done this, you must then forget about half of the "rules," because they don't apply. The most notable of these is the stuff about an unDIMENSIONED STRING VARIABLE always having a default value of ten bytes, and the computer always "reserving" that space. When it comes to WRITING to a DATA FILE, the only space "reserved" is two bytes plus the exact size of the STRING. Unlike numeric values, which all require the same "byte space" -- 5 bytes; each STRING value for each STRING variable could and probably will require an assortment of "byte spaces" when the values are written to a DATA FILE.

On the surface, if you fully understand the significance of the above sentence, this would seem to make RANDOM ACCESSing of DATA FILES, which contain STRING variables, an almost impossible task - how would one e er know the exact size of each and every RECORD? Consider this example:

RECORD #1	5	JONES	200
RECORD #2	19	SMITH	575
RECORD #3	154	CLARK	8000
RECORD #4	2784	ADAMS	2

Each RECORD in this FILE contains 17 bytes.

RECORD #1	5	JONES	200
	5 bytes +	5+2 bytes +	5 bytes = 17 bytes
RECORD #2	19	SMITH	575

an so on .....

The READ STATEMENT that would be used to RANDOM ACCESS this DATA FILE would be:

```
60 READ #0 %17*(J-1),A,B$,C
```

Which says:

SKIP 17 BYTES (J-1) TIMES AND THEN READ A,B$,C

Where J is the desired RECORD number.

That's simple enough -- but we had one thing going for us; all the names in the FILE were exactly the same number of characters - 5 bytes.

Consider this FILE:

RECORD #1	5	ROXANNE	200
RECORD #2	19	GREG	575
RECORD #3	154	KATHY	8000
RECORD #4	2784	ELIZABETH	2

RECORD #1	5		ROXANNE		200		
	5 bytes	+	7+2 bytes	+	5 bytes	=	19 bytes
RECORD #2	19		GREG		575		
	5 bytes	+	4+2 bytes	+	5 bytes	=	16 bytes
RECORD #3	154		KATHY		8000		
	5 bytes	+	5+2 bytes	+	5 bytes	=	17 bytes
RECORD #4	2784		ELIZABETH		2		
	5 bytes	+	9+2 bytes	+	5 bytes	=	21 bytes

Nothing short of a very complicated series of separate READ statements for each RECORD would allow us to RANDOM ACCESS this FILE as it is written. Yet, this FILE seems very typical, and common sense tells us that FILEs such as this are RANDOM ACCESSed a million times a day. "So how do they do it ?" you ask. Here's how....

I am sure that there are a covey of ways to do it, but the following method seems the most logical to me.

If we look at our second example FILE, we see that no name in it is greater than 9 bytes. If we are sure that all future DATA for the STRING variable will not exceed nine bytes, we then DIMENSION that STRING variable to nine -- DIM A$(9). We could just as easily have DIMENSIONed it to 10, or 22, or 200, or any size that we needed. Our expected DATA would have determined the required DIMENSION.

One would think that in view of the existence of the default DIMENSION of 10 for all unDIMENSIONed STRING variables, that the computer would automatically reserve 10 byte spaces, and we wouldn't have to bother with a DIMENSION statement, since we could just as easily used a default size of 10. Not so.... As I said earlier, all STRING variables occupy only the number of bytes for the characters in the STRING, plus 2 or 3 additional bytes for "disk format." If the STRING is less than 256 characters, it is 2 additional bytes; if greater than 256 characters, it is 3 additional bytes.

In order to RANDOM ACCESS a TYPE 3 DATA FILE which contains variable length STRING variables, one must devise a method of being able to determine in advance the size of each RECORD. The easiest way that I know to do this is to determine in advance the maximum size of the largest RECORD in the FILE, and then force the computer to make all the RECORDS in that FILE that size. This way you will know in advance the size of each and every RECORD, thus you will be able to provide the required "pointer instructions" for RANDOM ACCESS.

Again, considering our second example file if we make the first variable A, the second variable B$, and the third variable C, we would want to READ or WRITE A,B$,C for each RECORD in FILE$.

If we DIMENSION B$ to 9 bytes - DIM B$(9) - this would allow B$ to contain one to nine characters. That's not what we want, we want it to always contain exactly 9 characters (9 bytes). So we must create a second STRING variable - L$ - which will always be equal to 9 bytes. We do this by:

```
L$ = "........."
```

Where there are exactly nine "dots" between the quotation marks. This is the key to the whole thing - L$. Let's see how we will use the key to unlock the mechanism that will force the computer to always make B$ contain the required nine bytes. Thus, allowing us to determine the exact size of our RECORDS in our FILE$.

Consider this:

1st...By DIMENSIONing B$ to 9 --- DIM B$(9)
we allow B$ to contain 1 to 9 characters.

2rd...We have made L$="........."

3rd...We create yet another STRING variable B1$, which we also make DIM B1$(9). Our reason for DIMINSIONing B1$ to nine bytes is to limit its size to no more than 9 characters. If the STRING value assigned to B1$ is greater than 9 characters, the computer will automatically ignore any characters after the first nine.

4th...We will now make B1$ equal to the INPUT STRING variable - B$, plus our "made up" STRING variable - L$. Thus:

```
B1$ = B$ + L$
```

5th...If we INPUT JULIE for B$ then:

```
B1$ = B$ + L$

B1$ = JULIE + .........

B1$ = JULIE.........
```

Therefore, B1$ contains 14 characters

6th...Now, let's put it all together. Since DIM B1$(9), the computer will ignore all but the first nine characters, thus:

```
B1$=JULIE....
```

So, if we were to change L$ = "   " Where there are nine spaces between the quotation marks, then:

```
B1$ = JULIE
```

As far as the eye is concerned......

Now let's consider how we can use all these fabricated special conditions in a general sort of way. First we will CREATE a DATA FILE named FILE$. We do that from DOS by:

```
On  *  type in: CR FILE$ 10
On  *  type in: TY FILE$ 3
```

Then, we write a program to add DATA to this FILE$. This program will add DATA by SEQUENTIAL ACCESS. The program would be:

```
  5 DIM A$(11),B$(11),C$(11)
  6 DIM L$(11)
  7 DIM A1$(11),B1$(11),C1$(11)
 10 OPEN #0, "FILE$"
 20 INPUT "A$ = ",A$
 30 IF A$ = "0" THEN 120
 40 INPUT "B$ = ",B$
 50 INPUT "C$ = ", C$
 60 L$=".........."
 70 A1$ = A$ + L$
 80 B1$ = B$ + L$
 90 C1$ = C$ + L$
100 WRITE #0, A1$,B1$,C1$
110 GOTO 20
120 CLOSE #0
130 END
```

If we RUN we get:

```
A$ =   type in : FRANCE
B$ =   type in : ENGLAND
C$ =   type in : USA
A$ =   type in : NORWAY
B$ =   type in : CANADA
C$ =   type in : USSR
A$ =   type in : AUSTRALIA
B$ =   type in : MEXICO
C$ =   type in : DENMARK
A$ =   type in : POLAND
B$ =   type in : PERU
C$ =   type in : SWEDEN
A$ =   type in : 0
```

Now, let's quickly review what is happening when this program is executed.

LINE 5, 6, 7	These LINEs both set and limit the size of all STRING variables in the program
LINE 10	OPENs the FILE$
LINE 20, 40, 50	These LINEs assign values to A$,B$,and C$
LINE 30	This is the "escape" to terminate the program
LINE 60	This provides the "filler" to assure that all STRING variables contains eleven bytes
LINE 70, 80, 90	These LINEs assign values to A1$,B1$, and C1$ by taking the INPUT values and adding the "filler" to assure that each STRING variable contains exactly eleven bytes.
LINE 100	This WRITEs the DATA to FILE$
LINE 110	This sends the program back for more DATA
LINE 120	The program comes to this LINE when all DATA is complete and "0" is typed in as a response to the INPUT statement for A$. The FILE$ is then CLOSED and any DATA in the BUFFER is WRITTEN.

Let's now discuss how all of this relates to our specific example. In response to the three INPUT statements, we made:

```
A$ = FRANCE
B$ = ENGLAND
C$ = USA
```

The computer did as instructed and added L$ to each of the three STRING values. Thus:

```
A1$ = FRANCE...........
B1$ = ENGLAND..........
C1$ = USA..........
```

Or the INPUT value plus eleven "dots" for each.

However, since A1$,B1$, and C1$ have all been DIMENSIONed to eleven characters, they are then limited to only eleven characters, and the computer will ignore all but the first eleven characters. The resulting values then become:

```
A1$ = FRANCE.....
B1$ = ENGLAND....
C1$ = USA........
```

Now, each of these STRING variables contains exactly eleven characters (11 bytes), and we can "count" on that.

Each RECORD in our FILE$ now contains a pre-determined number of bytes, this allows us to RANDOM ACCESS and single RECORD or any single VARIABLE within that RECORD.

Our FILE$ looks like this:

```
RECORD #1       FRANCE.....ENGLAND....USA........
RECORD #2       NORWAY.....CANADA.....USSR.......
RECORD #3       AUSTRALIA..MEXICO.....DENMARK....
RECORD #4       POLAND.....PERU.......SWEDEN.....
```

Each RECORD contains 39 bytes.

```
RECORD #1       FRANCE.....ENGLAND....USA........

39 BYTES    =   11+2 bytes 11+2 bytes 11+2 bytes
```

Our READ statement to RANDOM ACCESS any RECORD in this FILE$ would be:

```
40 READ #0 %39*(J-1),A$,B$,C$
```

Note that there is no comma between READ #0 and the file pointer instructions, %39*(J-1). J is the desired RECORD number.

The program to READ our FILE$ would be:

```
10 OPEN #0, "FILE$"
20 INPUT "WHAT RECORD NUMBER DO YOU WANT : ",J
30 IF J = 0 THEN 80
40 READ #0 %39*(J-1),A$,B$,C$
50 !A$\!B$\!C$
60 CLOSE #0
70 GOTO 10
80 CLOSE #0
90 END
```

if we RUN we get:

```
WHAT RECORD NUMBER DO YOU WANT : type in: 3
AUSTRALIA..
MEXICO.....
DENMARK....
WHAT RECORD NUMBER DO YOU WANT : type in: 0
READY
```

Now let's suppose that all those "dots" are starting to bother you, and frankly you just don't want the things around. Now that you understand the reason for having them there, there is no further need to keep them there. Go back to the program that was used to put the DATA in the FILE$ and change LINE 60 to:

```
60 L$ = "           "
```

Where there are eleven blank spaces between the quotation marks. Then if you RUN you get:

```
WHAT RECORD NUMBER DO YOU WANT : type in: 3
AUSTRALIA
MEXICO
DENMARK
WHAT RECORD NUMBER DO YOU WANT : type in: 0
READY
```

Now, let's assume the inevitable happens. No sooner than we get all the world powers in reasonable order and on FILE$, we get a letter from the NEAR EAST telling us that if we don't include them they will "cut our water off." We've got to do it.

We READ our FILE$ and find that MEXICO is agreeable to a deletion, since they just discovered oil. That's the easy part, the tough part is changing our FILE$. We must write a program to allow us to select the desired RECORD, and then make the appropriate change for the desired VARIABLE, without upsetting the neighbors, AUSTRALIA and DENMARK. The required program would be:

```
 10 DIM K$(11),K1$(11),L$(11)
 20 INPUT "WHAT RECORD NUMBER DO YOU WANT : ",J
 30 IF J = 0 THEN 150
 40 INPUT "WHAT VARIABLE DO YOU WANT TO CHANGE : ",W$
 50 INPUT "WHAT DO YOU WANT TO CHANGE IT TO : ",K$
 60 IF W$ = "A$" THEN H = 0
 70 IF W$ = "B$" THEN H = 13
 80 IF W$ = "C$" THEN H = 26
 90 L$ = "..........."
100 K1$ = K$ + L$
110 OPEN #0, "FILE$"
120 WRITE #0 %39*(J-1)+H, K1$, NOENDMARK
130 CLOSE #0
140 GOTO 10
150 CLOSE #0
160 END
```

Again, I mention, if you don't want all those "dots" change LINE 90 to:

```
90 L$ = "           "
```

With eleven blank spaces between the quotation marks.

If we RUN we get:

```
WHAT RECORD NUMBER DO YOU WANT : type in: 3
WHAT VARIABLE DO YOU WANT TO CHANGE : type in: B$
WHAT DO YOU WANT TO CHANGE IT TO : type in: IRAN

WHAT RECORD NUMBER DO YOU WANT : type in: 0
READY
```

If you now go back and LOAD the program that was used to READ this FILE$ and RUN, you get:

```
WHAT RECORD NUMBER DO YOU WANT : type in: 3
AUSTRALIA..
IRAN.......
DENMARK....
```

We have now completed the longest and most demanding chapter in this book. There is much repetition here, because I feel that some of us need all the help we can get. If you understand all that I have written about RANDOM ACCESS of DATA FILEs, and it took you less than six months to understand it, then I can only assume one of two things; 1. You are considerably more intelligent than I, or 2. You had better instructive information available to you. I hope in your case it is the result of both.

34

DECIMAL AND HEXADECIMAL

Now, to what I consider more of the heavy stuff. On numerous occasions we have already used the computer's memory ADDRESS system to EXecute, JumP, ReaD, or LiFt a program. As we encountered them, I kept glossing over their explanation, or put it off till later, or on most occasions, not mentioning it at all. I didn't want to call it to your attention.

Each one of the COMMANDS was followed by a number. This number tells the computer where to get something or where to put something. There are two types of ADDRESS systems used, one is the DISK ADDRESS, the other is the computer's memory ADDRESS system.

The DISK ADDRESS has already been discussed in the chapter on RANDOM ACCESS. It has little or nothing to do with DECIMAL or HEXADECIMAL addresses, and should not be confused with them.

Consider the following examples which we have already used:

```
EX E900    RD 0 6000 32    JP 2A04    WR 32 5700 16
```

These are all COMMANDS directed at specific ADDRESSes in the memory of the computer. These are all HEXADECIMAL ADDRESSes. Up to now we have not used the DECIMAL ADDRESS.

There is a direct relationship between DECIMAL and HEXADECIMAL. All HEXADECIMAL ADDRESEs have an equivalent DECIMAL ADDRESS, and all DECIMAL ADDRESSes have an equivalent HEXADECIMAL ADDRESS. The relationship between them will be examined.

The HEXADECIMAL ADDRESS is the only address used by the Disk Operating System. All commands used in DOS which contain an ADDRESS must be in HEXADECIMAL. Note the above examples.

We have as yet have not encountered the use of the DECIMAL ADDRESS. The DECIMAL ADDRESS is used at the "byte access" level. In general, it is only used in the BASIC mode. Each "space" of memory has a specific DECIMAL ADDRESS. A 24K system has 24,576 DECIMAL ADDRESSes.

Some examples of the DECIMAL ADDRESS are:

```
10 FILL 52224,42

20 EXAM (57321)
```

The FILL statement will place a specific ASCII code in memory. For our example it will place 42 (*) in the computer's 52,224th "memory space."

The EXAM statement will EXAMine whatever is in the computer's memory at the designated DECIMAL ADDRESS. In our example it will look at the 57,321st "byte space." If it looked at the 52,224th "byte space" it would see 42.

Let's look at how the HEXADECIMAL system works.

Essentially the HEXADECIMAL system is no more than a alphabetic-numeric representation of the DECIMAL system, to the base SIXTEEN intead of TEN. For our purposes the smallest number would be zero and the largest number would be 65,535.

They would be represented by:

HEX	DEC		HEX	DEG
0000	0	$	FFFF	65535

Since HEXADECIMAL uses base sixteen, we need symbols for the "units" from 10 to 15, in addition to 1 to 9. We use the letters A to F to represent the digits 10 to 15.

Let's consider our highest example:

HEX	DEC
FFFF	65535

Starting at the left, the first number or letter tells you how many times to raise 16 to the third power.

Therefore:

F TIMES 16 CUBED

or:

F*16^3

The next number or letter tells you how many times to raise 16 to the second power.

Therefore:

F TIMES 16 SQUARED

or:

F*16^2

The next number or letter tells you how many times to raise 16 to the first power.

Therefore:

F TIMES 16

or:

F * 16

The last number or letter tells you how many times to raise 16 to the zero power. Since 16 to the zero power is one:

F TIMES 1 or F

The next question should be, "How do you know the value of the 'letter' in the HEXADECIMAL number?"

Here it is:

	1st	2rd	3rd	4th
	1	1	1	1
	!	!	!	!
	!	!	!	!
	9	9	9	9
	-	-	-	-
10 times	A	A	A	A
11 times	B	B	B	B
12 times	C	C	C	C
13 times	D	D	D	D
14 times	E	E	E	E
15 times	F	F	F	F

So again considering our highest example:

F F F F

```
1st    F  =  15 X 16^3  =  61440
2rd    F  =  15 X 16^2  =   3840
3rd    F  =  15 X 16^1  =    240
4th    F  =  15 X 16^0  =     15
                          ------
        HEX   F F F F   =  65535  DEC
```

Now let's try it with JP 2A04, which is the JumP command to get from DOS to a BASIC program in the computer's memory.

2 A 0 4

```
1st   2  =  2 X 16^3  =  2 X 4096  =  8192
2rd   A  = 10 X 16^2  = 10 X  256  =  2560
3rd   0  =  0 X 16^1  =  0 X   16  =     0
4th   4  =  4 X 16^0  =  4 X    1  =     4
                                     ------
         HEX   2  A  0  4          = 10756  DEC
```

Now for some practical considerations. Consider our example, RD 0 6000 32. What this tells the computer is to READ from a mini-disk STARTING at BLOCK 0 into the computer's memory STARTING AT HEX ADDRESS 6000 the NEXT 32 BLOCKS on the mini-disk.

The 6000 is the HEX address, since all DOS functions are done by HEXADECIMAL, telling the computer where to store the information that it is ReaDing off the mini-disk. In assigning this address several things must be considered. First one does not want to "store" the information where something else already is stored. In order to execute the ReaD command we had to load the Disk Operating System into the computer. DOS is at HEX 2000. So if we ReaD our mini-disk to 2000 - RD 0 2000 32 - we would overwrite DOS.

The next consideration is the size of what we are storing. We want to ReaD into the computer 32 BLOCKS. On a 24K system like mine, the highest HEX ADDRESS is 8000. With DOS loaded into the computer, the mini-disk can be ReaD into the computer anywhere between HEX 2000 plus 10 blocks for DOS and HEX 8000. Since each BLOCK is 256 bytes (16^2), we don't want to make our starting ADDRESS too high, or we will exceed the memory ADDRESSes of our system.

Let's look at the lowest possible starting address. We know that DOS is loaded into the computer starting at HEX ADDRESS 2000. We also know that DOS is 10 BLOCKS long - which is the same as 10 times 256 bytes long - which is the same as 10 times 16^2 long. Now, it's starting to look like one of the "factors" in the HEXADECIMAL system, and it is.

HEXADECIMAL equivalent of 10 X 16^2 = 0 A 0 0

If you add the DOS starting ADDRESS 2 0 0 0 to the number of BLOCKS required to store DOS - converted to HEXADECIMAL, you will end up with the DOS ending ADDRESS.

Thus:

```
DOS STARTING HEX ADDRESS  =  2 0 0 0
NUMBER OF BLOCKS IN HEX   =  0 A 0 0
                             ---------
DOS ENDING HEX ADDRESS    =  2 A 0 0
```

Which is the HEX starting ADDRESS for BASIC. Which makes sense, because BASIC is loaded into our system immediately following DOS in memory. However, for our example since we have not loaded BASIC this memory "space" is available.

Now back to the 32 BLOCKS that we want to ReaD into the computer's memory.

32 BLOCKS = 32 X 16^2 = HEX 2 0 0 0

If we add:

```
DOS ENDING HEX ADDRESS :  2 A 0 0
HEX VALUE OF 32 BLOCKS :  2 0 0 0
                         ---------
ENDING HEX ADDRESS     :  4 A 0 0
```

And since we can go up to HEX ADDRESS 8000, we will have no trouble ReaDing our 32 BLOCKS starting at HEX 2 A 0 0. So our ReaD command could have been:

RD 0 2A00 32

If we had BASIC in our computer, which has a starting HEX ADDRESS of 2 A 0 0, and is 45 BLOCKS long, our available memory for "storing" would be:

```
STARTING HEX ADDRESS FOR BASIC  : 2 A 0 0
HEX VALUE OF 45 BLOCKS          : 2 D 0 0
                                 ---------
ENDING HEX ADDRESS FOR BASIC      5 7 0 0
```

Our ReaD command could have been:

RD 0 5700 32

And our ending HEX ADDRESS would be:

7 7 0 0

(5 7 0 0 + 2 0 0 0 = 7 7 0 0)

Which is still less than our maximum HEX ADDRESS 8 0 0 0.

But we choose a convenient number to remember -- 6000.

So:

```
STARTING HEX ADDRESS  :  6 0 0 0
HEX VALUE OF 32 BLOCKS:  2 0 0 0
                         ---------
ENDING HEX ADDRESS       8 0 0 0
```

This equals our maximum HEX address of 8 0 0 0, so we would just be able to "store" 32 BLOCKS starting at HEX 6 0 0 0.

Let's look at the arrangement of memory in the computer. Unique to this system (SOL-20 with NORTH STAR BASIC) is the fact that the first 8K of memory is not used, and therefore the first 8 times 1024 of DECIMAL or HEX addresses are not used. The reason for this is beyond the scope of this book. However, that does not mean that you have lost 8K of available memory. It's all there, it's just at a different ADDRESS.

This is the way the computer's memory bank looks:

```
 0
 !  No memory
 !
8K
 !   1st 8K
 !   memory
16K
 !   2rd 8K
 !   memory
24K
 !   3rd 8K
 !   memory
32K
```

This arrangement of memory causes no problems, but you must make suitable adjustments in figuring the HEX or DECIMAL ADDRESSes.

The next thing to remember is that "K" is not equal to 1000, but that "K" is equal to 1024. Therefore a 24K system has:

24K = 24 X 1024 = 24576 - 1 bytes

Or 24575 byte ADDRESSes. The reason for the "- 1" is because the computer starts counting at zero instead of one.

Here is another "map" of the computer's memory:

```
                         DECIMAL
                         ADDRESS
                            0
                            1      1st 8 K
                            1     NO MEMORY
                            1
      ----------------------8191 = 8 X 1024 minus 1  --------
      1                     1                               1
      1                     1                               i
      1                     1                               1
      1         ------------8192  =  2 0 0 0  HEX           1
      1         1           1                               1
      1         1           1          STARTING DOS         1
      1    2,560 bytes      1                               1
      1         1           1               10 BLOCKS       1
      1         1           1               2,560 bytes     1
      1         1           1                               1
      1         -----------10751        END DOS             1
      1                     1                               1
      1                     1                               1
      1                     1                               1
      1         -----------10752  =  2 A 0 0  HEX           1
      1         1           1                               1
      1         1           1           STARTING BASIC      1
 24K  1         1           1                               1
24,576 1   11,520 bytes     1                45 BLOCKS      1
bytes 1         1           1               11,520 bytes    1
      1         1           1                (45 X 256)     1
      1         1           1                               1
      1         -----------22271           END BASIC        1
      1                     1                               1
      1                     1                               1
      1                     1                               1
      1         -----------22272  =  5 7 0 0  HEX           1
      1         1           1                               1
      1    10,495 bytes     1          START  FREE          1
      1         1           1             MEMORY            1
      1         1           1                               1
      1         -----------32767   END OF MEMORY  -----------
      1                     1
      1                     1
      ---------------------32768  =  8 0 0 0  HEX
```

If we can get from HEX to DECIMAL, then we ought to be able to go the oppsite way - DECIMAL to HEX. We can, but it is a little different.

Here is how:

STARTING : 22310 DECIMAL

....

22310 divided by 16^3 = 5 + remainder: 1830

1830 divided by 16^2 = 7 + remainder: 38

38 divided by 16^1 = 2 + remainder: 6

6 divided by 16^0 = 6

. . . .

HEXADECIMAL equivalent for a DECIMAL number is the quotients from the above calculation:

DECIMAL 22310 = HEXADECIMAL 5726

If any of the quotients had been between 10 and 15, we would substitute the HEX letter for the value: 10=A, 11=B,...,15=F. If a quotient is more than 15 it means that we didn't start with a high enough power of 16.

. . . .

That is all that one really needs to know about the relationship between DECIMAL and HEXADECIMAL at this time. To be completely honest with you, you probably don't really "need" to know this much. Most of the HEX ADDRESSes that you will use are already provided and by the time you start working with DECIMAL ADDRESSes, byte access, you will have long passed the need for this book. On the other hand, understanding this chapter will enable you to better understand why and what you are doing.

I have included a program in the back of the book which will convert HEX to DECIMAL or DECIMAL to HEX, which further precludes the necessity for this chapter. In spite of all the reasons not to become familiar with the subject at hand, I highly recommend the converse.......

35

SECRETS

I honestly wish that this could be the longest chapter in the book, but it rivals the shortest.

A secret, according to the Random House Dictionary, College Edition, is "a method, formula, plan, etc., known only to the initiated or the few." I have found that a lot of the "general knowledge" about the finer points of how to operate and program these microcomputers qualifies as a secret by the above definition.

My experience has been like dealing with the little old lady who gives you her favorite recipe, but "forgets" to include one or two of the most significant ingredients. When you try her recipe, what you get is all right, but never as good as hers --- and you never really understand why.

I have found "computer people" to be a lot like the "little old lady." They seem to always make their computers do more than you can make yours do. They always say they've told you everything, but if you watch and listen real close, they will always do something they "forgot" to tell you about.

After two six-paks of beverage, I finally got one of those "people" talking. However, in order to keep him drinking (and talking), I had to drink along. This proved to be a disaster. I don't remember much of what he said, and my notes weren't as great the next day as I thought they were that night. From my early evening notes I was able to glean some "secrets," though not as many as are still out there.

Here's what I got:

1. How to change the display and INPUT speed of the Sol computer.

   On PROMPT or READY type in: FILL 51211,45

   The number after the FILL 51211, can be any number between 0 and 255, zero being the fastest speed and 255 being the slowest speed. The display stays at this speed until

changed again by this statement. It can also be included in a program so that it can "print" on the screen at different speeds depending on what's going on in the program.

2. The video portion of the Sol computer's memory is between 52224 and 53247. If you want the computer to PRINT any character that you choose, in any position of the monitor, you would type:

```
FILL 52224,42
```

Which would put the character * somewhere on the screen. (42 is the ASCII code for *) I think he said the top left corner if the screen was cleared to start with (see below for how to clear the screen).

3. To clear the monitor of all characters, you would type:

```
! CHR$(11)
```

This one was obvious, after he told me it does the same thing as the CLEAR key on the keyboard does when we're in the START-UP mode.

4. I got this program off the table cloth the next day:

```
 10 ! CHR$(11)
 20 !"THIS WILL PLOT ANY CHARACTER YOU WANT"
 30 !"ANYWHERE ON THE SCREEN."
 40 !
 50 !"IS THE CHARACTER YOU WANT ON THE KEYBOARD ?"
 60 A$ = INCHAR$(252)
 70 IF A$ = "Y" THEN 110
 80 INPUT "ENTER THE ASCII CODE : ", C
 90 A$ = CHR$(C)
100 GOTO 130
110 ! "WHAT CHARACTER ?"
```

```
120 A$ = INCHAR$(252)
130 !"INVERSE VIDEO ?  (Y OR N)
140 X$ = INCHAR$(252)
150 IF X$ = "Y" THEN A$ = CHR$(ASC(A$)+128)
160 INPUT "LINE (0 TO 15) : ",L
170 INPUT "CHARACTER POSITION (0 TO 63) : ",K
175 ! CHR$(11)
180 FILL 52224 +64*L + K, ASC(A$)
190 GOTO 150
```

The next day, I put it in the computer, and it provided "food for thought" all day. (I think some of it only works with the "NEW" BASIC, and just on the Sol).

5. To turn your Sol computer into just a terminal to work with another computer, you:

   On > type in: TE

   I didn't know this, and I've never had a need for this information, but I guess it's good to know.

6. To make your computer PRINT on a printer, instead of your monitor, you would:

   On PROMPT or READY type in: FILL 51207,1

   To stop your computer from PRINTing on a printer, and display on the monitor, you would:

   On PROMPT or READY type in: FILL 51207,0

   Everybody seemed to know this, but me. I also found out that this only applies to a "serial" printer. If you have a "parallel" printer, use a 2 instead of a 1 after the FILL statement. This one will also work from inside a program.

7. How do you make the computer print a quotation mark in a PRINT statement?

```
60 !CHR$(34),"COMPUTERS ARE FUN",CHR$(34)
```

That's how......

8. Here's how to change DOS such that you can go directly from START UP to READY by only typing in EX E900 and then pressing the RETURN key. The computer will automatically GO BASIC and LOAD it for you.

1st LOAD DOS . . . on > type in: EX E900

2rd LOAD BASIC . . on * type in: GO BASIC

3rd Return to DOS. on READY type in: BYE

4th LOAD DOS at 3000 . . on * type in: LF DOS 3000

5th Go to START UP . . .press UPPER CASE and REPEAT

6th Change the "FLAG" byte 38E5 (HEX) from 01 to 00:

> On > type in: EN 38E5
> Press the RETURN key.
> on : type in: 00/
> Press the RETURN key.

7th Go to BASIC . . . on > type in: EN 2A04

8th Go to DOS . . . . on READY type in: BYE

9th Put in "new" INITIALIZED mini-disk.

10th SAVE modified DOS . . on * type in: SF DOS 3000
11th SAVE BASIC . . . on * type in: SF BASIC 2A00

For some versions of DOS this may not be sufficient. If the above didn't work for you, then that means that the GO

BASIC instructions are not a part of your DOS. So let's put them there. To do this, after step 6, in the above procedure, (you can repeat the whole thing) you should:

On > type in: EN 37C0

Press the RETURN key.

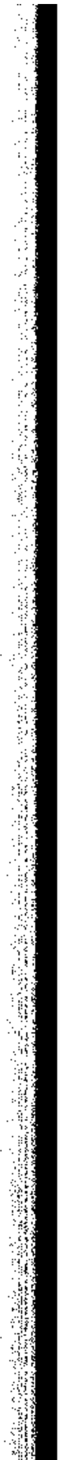
On : type in: 47 4F 20 42 41 53 59 43 0D/

Press the RETURN key.

Proceed to STEP 7

All those funny numbers say "GO BASIC" in HEX. The "0D" is the HEX representation for a carriage return.

There were a lot more "secrets" disclosed that night, as we talked, and drank, till dawn. I just can't remember them. I would highly recommend taking your friendly computer store person out and buying him/her a few beverages. I would further recommend, that you write and he/she drink. HOWEVER, under no circumstances, invite two or more "computer people" out at one time, if your objective is to learn something. If you can get a "computer type" to sit down with you at your computer, you can also discover a lot of the "secrets" from them.



36

THE NEW BASIC

There is always something to be gained from learning the fundamentals, but it is not always necessary to know them to be able to function. The NEW BASIC is exactly like the "old" BASIC execpt that it is different. It does everything that the "old" BASIC does, for the most part, exactly the same. It also does a few new things. Almost anything that you learn about the "old" BASIC is applicable to the NEW BASIC.

The NEW BASIC is North Star's Release 4 (June, 1978). This book was complete and ready for the publisher, when I received a copy of the NEW BASIC. My first impulse was not to include it. However, after working with it a few days, I decided that the NEW BASIC had something to offer that you should know about.

The biggest changes are the automation of the most used "functions" relating to SAVING and LISTING files. These make life a little eaiser and eliminates a great deal of JumPing and BYEing. Now you can do it all from BASIC mode. There are probably a lot more advantages than I can evaluate, because I don't fully understand everything that the "old" BASIC can do yet. One thing that I know that hasn't changed is the person that writes the manuals for North Star. I honestly could not understand most of it, but then I may have been lacking some "familiarity with some version of BASIC."

After much reading and much "trial and error" time with my computer, I finally figured out the following:

1. The FREE command now has a partner. As you remember, the FREE command, from BASIC mode, will tell you how much FREE (available) memory, in bytes, you have left in the computer.

   Now, with the new BASIC you have the PSIZE command. The PSIZE command also operates in the BASIC mode. When executed the PSIZE command will tell you how much memory, in blocks, that you have used for the program in the computer.

   Thus FREE tells you how much you've got left in bytes, and PSIZE tells you how much you've used in blocks. (block = 256 bytes)

   If you have written a program and you want to

know how many blocks of disk space would be required to SAVE your program, you would:

on READY or CURSOR type in: FSIZE

and the computer would respond by telling you how many blocks would be required.

2. The new function - NSAVE - is a welcome addition. From the BASIC mode it will allow you to SAVE the program that you have in the computer without having to go to the DOS mode, CReate a file, TYpe it, size it, and then back to BASIC to SAVE it. The NSAVE command does all of that for you automatically. Again, all the "old" BASIC operations to do this, still work too.

3. The CAT command will allow you to LIst the names of all the files that you have on any mini-disk in the disk reader from the BASIC mode. Heretofore you had to go to the DOS mode first. Now you can use CAT from the BASIC mode and LI from the DOS mode.

4. The next new feature is the AUTO command. This command will cause the computer to automatically write the LINE NUMBERS for you, when you are writing a BASIC program. When this command is executed, the computer will generate a new LINE number, everytime you press the RETURN key.

   You can specify the starting LINE number and the size of the increment, if you want. Or, if you don't want, the computer will assign the default starting LINE number of 10, and the default increment of 10. All you do is:

   on prompt or READY : type in: AUTO

Thus:

```
10. . . . . . . . .
20. . . . . . . . .
30. . . . . . . . .
```

If you want to start at a LINE number other than 10, then:

on prompt or READY : type in: AUTO 100

This will start with LINE 100, and each subsequent LINE number will increase by the default increment of 10.

Thus:

```
100. . . . . . . . .
110. . . . . . . . .
120. . . . . . . . .
```

If you want to start at LINE 100 and increment by 100, you would then:

on prompt or READY : type in: AUTO 100,100

And you would get:

```
100. . . . . . . . .
200. . . . . . . . .
300. . . . . . . . .
```

Two things that make the AUTO command awkward to use for me:

a. There is no space automatically provided between the LINE number and the LINE statement, so you have to remember to put it there if you want it. It's not necessary, and it does take up memory. Still, for beginners and programmers who like to be able to find the right LINE number in a hurry, I recommend the "space."

b. I'm not a good enough programmer to write a program without having to "insert" LINEs as I go along. Thus, I always start out using the AUTO command, but very quickly abandon it, because it's more trouble to stop and restart it than it is to just put in the LINE numbers. But, I always start out using it.

I would recommend for the next version of NEW BASIC to rectify these two detractants, by making the space between LINE number and LINE statement optional, and showing the computer owner how to personalize the AUTO command, if possible. Also, by including an "escape" from the AUTO command, which would allow one to exit it, use whatever LINE numbers he wants, and then re-enter the AUTO, such that it would retain its "count."

5. The APPEND command will probably not be one of the most used features of the NEW BASIC, but it's good to have around. This command allows one to LOAD a program from a mini-disk on to a program that is already in the computer.

   The one condition required to use the APPEND command, is also the only draw-back I see to the use of the APPEND command. The last LINE number of the program in the computer must be lower than the first LINE number of the program that you are LOADing into the computer.

   Thus if you have:

```
        PROG-1

    10. . . . . . .
    20. . . . . . .
    30. . . . . . .
    40. . . . . . .


        PROG-2

    10. . . . . . . .
    20. . . . . . . .
    30. . . . . . . .
    40. . . . . . . .
```

Where, PROG-1 is in the computer and PROG-2 is on a mini-disk, and you wish to APPEND PROG-2 to the end of PROG-1, this can not be done without changing all the LINE numbers of PROG-2, such that the first LINE number is greater than LINE 40.

If PROG-1 is in the computer and you want to add PROG-2 to the end of it - APPEND PROG-2. You do this:

on prompt or READY : type in: APPEND PROG-2

If in the NEW NEW BASIC they include an automatic LINE NUMBER CHANGER, it would be great, without it - it's only good.

6. The new BASIC provides a very flexible method for RENumbering any program. You may select the beginning value for the first LINE of the program to be RENumbered. If none is selected the default starting LINE is 10. In addition to the starting LINE NUMBER, you may also select the increment value for subsequent LINE NUMBERS. If none is selected, the default value of increments of 10 is automatically assigned. Here' how:

   To RENumber any program in the computer all that do is type in REN and press press the return key, in the BASIC mode. This will make the first LINE of the program LINE 10, and all subsequent LINE NUMBERS will increase in increments of 10.

   10,20,30,........65510,65520,65530

   If we wanted or first LINE NUMBER to be 100 and each subsequent LINE NUMBER to increase in increments of 10, we would then type:

   REN 100

   Then press the RETURN key. We would get:

   100,110,120,.......56610,65520,65530

If we wanted our first LINE NUMBER to begin with LINE NUMBER 1000, and each subsequent LINE NUMBER to increase by 100. We would:

REN 1000,100

After we pressed the RETURN key and LISTed the program we would find the desired sequence of LINE NUMBERs:

1000,1100,1200,......65300,65400,65500

7. Another change is the new entry point to BASIC from DOS or START UP. For the old BASIC it was HEX 2A04. If you were in DOS mode and you wanted to get back to BASIC mode, you would:

on * type in: JP 2A04

From START UP you would:

on > type in: EX 2A04

For the new BASIC both of these commands still work and are both useful. But, another entry address has been provided that retains the value of all the variables. This new entry address is HEX 2A14. Thus, if you are RUNning a BASIC program and STOP it, and go to DOS mode, and then want to go back to the program, keeping all the values, you would:

on * type in: JP 2A14

If you want to re-enter the program with all the values reset, then:

on * type in: JP 2A04

as always....

8. The new BASIC has also corrected what I considered to be a major irritation in the old BASIC. The DELete command has been "fixed" so that once used on large blocks of LINE numbers, that these DELeted LINEs are usable again. As you may remember, I cautioned the reader about this "flaw" in an earlier chapter.

9. While we are on the subject of deletions, I will take this opportunity to mention that the old DE (DEletion) command, used to DElete files from the mini-disk, has been supplemented with DESTROY in the new BASIC.

   "old"   DE PROBLEM

   "new"   DESTROY "PROBLEM"

   Not only has the command been changed, but so has the command mode. The DESTROY command operates in the BASIC mode. Thus:

   on READY or CURSOR: type in: DESTROY "OLD"

   But like everything else, "the more things change, the more they remain the same." That is to say, the "old" way still works too. So if you're in the BASIC mode and want to DESTROY a FILE you can. If you are in the DOS mode and you want to DElete a FILE you still can.

10. The LIST command for LISTing a BASIC program in the memory of the computer has changed somewhat. Actually the commands have remained the same, but their meaning changed.

    The "old"   LIST 250

    Would LIST a program from LINE 250 to END.

The "new" LIST 250

will LIST LINE 250 only.

The "old" LIST 250,

would LIST only LINE 250

And:

The "new" LIST 250,

will LIST the entire program from LINE 250 to the END.

Everything else remains the same with regard to LISTing.

11. Another item of change that I will go into is the COMPACT command. In the old BASIC, if you had a mini-disk which you had DEleted several of the files from, and you wanted to "move" everything to the front of the mini-disk, you would put the computer in the DOS mode, insert the desired mini-disk, and:

on * type in: CO

In the new BASIC, you first load the COMPACT program by:

on * type in: GO COMPACT

Then insert the desired mini-disk into the disk-reader and, type in: 1.

There are several other features to the new BASIC which I either have not used enough to write about, or that I tried to use, but as yet do not fully understand.

In general, the new BASIC is an improvement over the old BASIC. If given the choice, I would recommend starting with the new BASIC. If you already have the old BASIC, I would recommend getting the new one when you get a chance. However, it's not a major advancement in the computer industry, it's just an improved version of what you already have, and what you already have works pretty good. I'm told that 99% of programs written with the old BASIC will work with no changes with the new BASIC. So I would not make a mad rush to your neighborhood computer store to get it, but if you are there - pick it up.

The end...but really just the beginning....

programs

```
10 REM  THIS PROGRAM PRINTS A LIST OF ALL THE POSSIBLE VARIABLES
20 REM
30 ! " IF YOU WANT A LIST OF STRING VARIABLES TYPE IN: 1"
40 !
50 ! " IF YOU WANT A LIST OF NUMERIC VARIABLES TYPE IN: 2"
60 !
70 INPUT "WHICH DO YOU WANT  1  or  2 : ",V
80 ON V GOTO 90,220
90 DIM A$(26), B$(26)
100 ! "         S T R I N G   V A R I A B L E   L I S T"
110 !
120 FOR I = 1 TO 26
130 FOR J = 1 TO 11
140 LET A$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
150 LET B$ = " 0123456789"
160 T$ = "$"
170 ! A$(I,I), B$(J,J), T$, "   ",
180 NEXT J
190 !
200 NEXT I
210 END
220 DIM A$(26), B$(11)
230 ! "        N U M E R I C A L   V A R I A B L E   L I S T"
240 !
250 FOR I = 1 TO 26
260 FOR J = 1 TO 11
270 LET A$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
280 LET B$ = " 0123456789"
290 ! A$(I,I), B$(J,J), "    ",
300 NEXT J
310 !
320 NEXT I
330 END
```

```
1 REM  THIS PROGRAM READS ANYTHING THAT IS READ INTO THE
2 REM  MEMOR OF THE COMPUTER FROM A MINI-DISK
3 !
10 REM INPUT VALUES\ S=24576 ----E= 32768
20 REM LOAD MENORY  ====  ON # RD '#' 6000 32
30 REM  WHERE # =0, 32 , 64 , 96 , -----  350
40 REM THEN  *  JP 2A04
50 REM THEN    RUN
60 REM EACH BLOCK IS 256 BITS \ IF # =1 THEN E=24576 +256
70 REM  S  &  E  ARE SET FOR 32 BLOCKS
80 S=24576
90 E=32768
100 FOR N=S TO E
110 A=EXAM(N)
120 IF A>128 THEN A=A-128
130 IF A<32 THEN A=46
140 !CHR$(A),
150 NEXT N
```

```
10 REM  THIS IS A SIMPLE CHECK BOOK BALANCING PROGRAM
20 INPUT " STARTING BALANCE FROM BANK STATEMENT   :  ", A
30 !
40 INPUT " HAVE YOU MADE A DEPOSIT  ?  :  ", A$
50 IF A$ = "NO" THEN 130
60!
70 INPUT " WHAT WAS THE AMOUNT OF THE DEPOSIT  :  ", E
80 F= F + E
90 K = K + 1
100 !
110 INPUT " DID YOU MAKE ANOTHER DEPOSIT  ?  :  ", A1$
120 IF A1$ = "YES" THEN 70
130!
140 INPUT " AMOUNT OF CHECK WRITTEN   :  ", B
150 C = C + 1
160 D = D + B
170!
180 INPUT " HAVE YOU WRITTEN ANOTHER CHECK  ?   :  " , Y$
190 IF Y$ = "YES" THEN 140
200 G = A + F - D
210!\!\!"*********************************************"
220!
230 ! " YOU HAVE WRITTEN  ",C,"  CHECKS  : TOTAL  ",$$C10F2,D
240 ! " YOU HAVE MADE  ", K, "  DEPOSITS  : TOTAL  ",$$C10F2,F
250!
260 H = SGN(G)
270 IF H = 1 THEN 320
280 IF H = -1 THEN 350
290! "         YOU ARE  N O T   O V E R D R A W N        "
300!"  BUT : THERE IS NO MONEY LEFT IN YOUR CHECKING ACCOUNT  "
310 GOTO 380
320! "    Y O U  S T I L L  H A V E  M O N E Y  L E F T  "
330 ! " THERE IS  ", $$C10F2,G,"  IN YOUR CHECKING ACCOUNT  "
340 GOTO 380
350 !"         Y O U  A R E  O V E R D R A W N   "
360 !" YOU MUST DEPOSIT  ",$$C10F2, ABS(G),"  TO COVER CHECKS",
370! " ALREADY WRITTEN "
380 END
```

```
1 REM  THIS PROGRAM WILL READ AND PRINT THE CONTENTS OF ANY
2 REM  TYPE THREE FILE
10 DIM A$(1000)
20 INPUT "FILE NAME :   ", B$
30 OPEN #0, B$
40 IF TYP(0)=0 THEN END
50 IF TYP(0)=1 THEN 90
60 READ #0,N
70 ! TAB(6),N
80 GOTO 40
90 READ #0, A$
100 ! TAB(15), A$
110 GOTO 40
```

```
10 ! CHR$(139)
20 FOR N=1 TO 4 \ ! CHR$(19), \ NEXT N
30 READ X,Y
40 IF X=0 THEN 230
50 IF X=0 THEN 230
60 FILL(52224 + X + 64* Y ) , 160
70 GOTO 30
80 REM FIRST LEFT
90 DATA 18,13,18,12,18,11,18,10,18,9,18,8,18,7,18,6
100 REM FIRST BOTTOM
110 DATA 19,13,20,13,21,13,22,13,23,13,24,13,25,13,26,13,27,13
120 DATA 28,13,29,13,30,13
130 DATA 41,13,31,13,32,13,33,13,34,13,35,13,36,13,37,13,38,13
140 DATA 39,13,40,13
150 REM FIRST RIGHT SIDE
160 DATA 41,12,41,11,41,10,41,9,41,8,41,7,41,6
170 DATA 40,12,40,11,40,10,40,9,40,8,40,7,40,6
180 DATA 19,13,19,12,19,11,19,10,19,9,19,8,19,7,19,6,40,13
190 REM FIRST TOP
200 DATA 19,6,40,6,20,6,21,6,22,6,23,6,24,6,25,6,26,6,27,6,28,6
210 DATA 29,6,30,6,31,6,32,6,33,6,34,6,35,6,36,6,37,6,38,6,39,6
220 DATA 41,6,0,0
230 RESTORE
240 FOR N= 1 TO 1000\ NEXT N
250 GOTO 10
260 END
```

```
10 ! " IF YOU KNOW THE CHARACTER AND WANT THE ASCII CODE TYPE IN: 1"
20 !
30 !"IF YOU KNOW THE ASCII CODE AND WANT THE CHARACTER TYPE IN: 2"
40 !
50 INPUT "WHAT DO YOU WANT  1  or  2 : ",V
60 !
70 ON V GOTO 150, 80
80 INPUT " WHAT IS THE CHARACTER FOR ASCII CODE NUMBER: ",A
90 !
100 !
110 !"THE CHARACTER FOR ASCII CODE ",A," IS : ",CHR$(A)
120 !
130 !
140 GOTO 10
150 INPUT "WHAT CHARACTER DO YOU WANT THE ASCII CODE FOR: ",A$
160 !
170 !
180 !"THE ASCII CODE FOR ", A$, " IS: ",ASC(A$)
190 !
200 !
210 GOTO 10
```

```
10 REM  THIS PROGRAM MATCHES THREE TYPES OF DATA AGAINST STORED
20 REM  DATA.  THE INPUT IS 21 CHARACTERS EITHER\ + , - , *
30 REM  THE + CAN REPRESENT POSITIVE, THE - CAN HEPRESENT NEGATIVE,
40 REM  THE * REPRESENTS A VARIABLE REACTION WHICH CAN BE + OR -.
50 REM  AS THE PROGRAM IS WRITTEN THE INPUT MUST BE 21 CHARACTERS
60 DIM R$(21), R1$(21)
70 INPUT "WHAT ARE YOUR 21 REACTIONS : ",R$
80 IF LEN(R$) <> 21 THEN 70
90 READ R1$,C
100 IF C = 0 THEN 180
110 FOR N = 1 TO 21
120 IF R1$(N,N) = "*" THEN 150
130 IF R$(N,N) = "*" THEN 150
140 IF R$(N,N) <> R1$(N,N) THEN EXIT 90
150 NEXT N
160 !" DATA MATCH FOR NUMBER : ",C
170 GOTO 90
180 RESTORE
190 ! "    END OF DATA"
200 GOTO 70
210 REM  YOUR DATA STATEMENTS MAY BE CHANGED TO SUIT YOUR
220 REM  OWN REQUIREMENTS  -  THE REACTIONS MUST BE
230 REM  ENCLOSED IN QUOTATION MARKS.
240 DATA "+++---+++---+++---+++",1
250 DATA "---+++---+++---+++---",2
260 DATA "***+++***+++***+++***",3
270 DATA "***---***---***---***",4
280 DATA "+*-+*-+*-+*-+*-+*-+*-",5
290 DATA "---------------------",6
300 DATA "+++++++++++++++++++++",7
310 DATA "*********************",8
320 DATA "++++++++++*----------",9
330 DATA "----------*++++++++++",10
340 REM   YOUR LAST DATA STATEMENT MUST BE\
350 DATA "000000000000000000000",0
```

```
1 REM  YOU MUST INPUT FOUR VALUES FOR HEXADECIMAL VALUES.
2 REM  FF  MUST BE INPUT\ DOFF
10 DIM A$(15)
20!
30!
31!
35 !"DO YOU WANT #1-DECIMAL TO HEX or #2 - HEX TO DECIMAL ?"
40 INPUT " 1 OR 2 ",L
45!
50 ON L GOTO 70, 270
60!\!
70 INPUT "WHAT DECIMAL NUMBER DO YOU WANT A HEX VALUE FOR : ",A
80 IF A>65535 THEN 240
90!
100 A$="123456789ABCDEF"
110 FOR Y=1 TO 4
120 X(1)= INT ( A/16^3)
130 X(2) = INT ((A - (16^3*INT(A/16^3)))/16^2)
140 X(3) =INT (   ( A -(INT(X(1))*4096 ) -(INT(X(2))*256 ))/16)
150 X(4) = A - (16*INT(A/16))
160 IF Y=1 THEN ! TAB(39), "HEX VALUE = ",
170 IF X(Y)=0 THEN 210
180 !TAB(51),A$(X(Y),X(Y)),
190 NEXT Y
200 GOTO 30
210 IF X(Y-1)>0 THEN ! "0",
220 NEXT Y
230 GOTO 30
240!\!
250 !" THIS PROGRAM ONLY TAKES NUMBERS UP TO 65536"
260 GOTO 30
270 INPUT "WHAT HEX VALUE DO YOU WANT THE DECIMAL NUMBER :",B$
280 K=0\J=0\T=0\S=0\B1$="0"
290 FOR J = 1 TO 4
300 K=K+1
310 LET B1$=B$(J,J)
320 T=0
330 IF B1$="A" THEN T=10
340 IF B1$="B" THEN T=11
350 IF B1$="C" THEN T=12
360 IF B1$="D" THEN T=13
370 IF B1$="E" THEN T=14
380 IF B1$="F" THEN T=15
390 IF T>9 THEN 410
400 T=VAL(B1$)
410 IF K=1 THEN M=T*16^3
420 IF K=2 THEN M=T*16^2
430 IF K=3 THEN M=T*16^1
440 IF K=4 THEN M=T
450 S = S + M
460 NEXT J
470!TAB(22),"THE DECIMAL VALUE IS : ",S
480 !
490 GOTO 30
```

```
1 REM  THIS IS A GAME OF GUESS THE NUMBER THAT THE COMPUTER
2 REM  IS THINKING OF.  THE COMPUTER WILL TELL YOU IF YOUR
3 REM  GUESS IS TOO HIGH OR TOO LOW, YOU THEN TRY ANOTHER
4 REM  GUESS.
10 A=0
20 X=INT(RND(0)*100+1)
30 A=A+1
40 INPUT " WHAT DO YOU THINK THE NUMBER IS ?    " ,Y
50 IF X>Y THEN 80
60 IF X<Y THEN 120
70 IF X=Y THEN 150
80!" YOUR NUMBER IS TOO                    LOW"
90!
100!
110 GOTO 30
120!" YOUR NUMBER IS TOO                    HIGH"
130!
140GOTO 30
150!
160!
170!"*************************************************************"
180!
190!"    Y O U   A R E   C O R R E C T  -  Y O U   G O T   I T  !
210!"*************************************************************"
220!
230!" IT TOOK YOU THIS MANY TUNRS TO GET IT  :  ",A
240!
250!
260 GOTO 10
```

```
1 REM  THIS AN INVOICE TOTALING PROGRAM
10 INPUT "UNIT COST: ",F
15 INPUT "HOW MANY OF THIS ITEM: ",H
16 C = F * H
20 ! "IS THIS A TAXABLE ITEM ?"
30 INPUT "YES OR NO: ",Y$
40 IF Y$ = "YES" THEN GOSUB 140
50 R = R + C
60 INPUT "DO YOU HAVE ANOTHER ITEM ?  YES OR NO: ",D$
70 IF D$ = "YES" THEN 10
80 ! %#$10F2
90 ! "TOTAL COST OF ALL ITEMS: ",R
100 ! "TOTAL COST OF TAXABLE ITEMS: ",B
110 ! "TOTAL AMOUNT OF TAX: ", .06*B
120 ! "TOTAL OF ALL ITEMS PLUS TAX: ", R+.06*B
130 END
140 B = B + C
150 RETURN
```

```
1 REM   THIS IS AN INVOICE TOTALING PROGRAM WITH ACCOUNTING
2 REM   OF ALL THE INVOICES, PLUS DISCOUNT CODES .
10 I=0\D=0\J=0\H=0\K=0\S=0\T=0\Q=0\R=0
20 N=N+1
30 INPUT "IS THIS ACCOUNT TAXABLE ?  : ",A$
40 IF A$ = "YES" THEN T=.05
50 I = I+1
60!
70 INPUT " WHAT IS THE LIST PRICE OF THE ITEM ? :  ",P
80!
90 INPUT " HOW MANY ARE TO BE SHIPPED ON THIS INVOICE : ",Q
100!
110 R=R+Q
120 INPUT "WHICH DISCOUNT CODE  1 , 2 , OR  3  : ", D
130!
140 GOSUB 330
150 INPUT "DO YOU HAVE ANOTHER ITEM  : ", B$
160 IF B$ = "YES" THEN 50
170!\!\!"---------------------------------------------"
180!
190 ! " TOTAL NUMBER OF LINE ITEMS FOR THIS INVOICE :  ",%#,I
200 ! " TOTAL NUMBER OF UNITS FOR THIS INVOICE  :  ",R
210 ! " TOTAL LIST PRICE ALL PRODUCT ON INVOICE :  ",%#$10F2,H
220 ! " TOTAL DISCOUNT PRICE ALL PRODUCT        :  ",K
230 ! " TOTAL TAX ON THIS INVOICE               :  ",K*T
240 ! " TOTAL AMOUNT OF THIS INVOICE            :  ",K+(K*T)
250 M=M+(K+(K*T))
260!
270 INPUT "DO YOU HAVE ANOTHER INVOICE  : ", C$
280 IF C$="YES" THEN 10
290!
300! " TOTAL NUMBER OF INVOICES PREPARED      :  ", %#,N
310! " TOTAL AMOUNT OF ALL INVOICES PREPARED  :  ", %$10F2,M
320 END
330 ON D GOTO 340,360,380
340 J=P
350 GOTO 390
360 J=P*.9
370 GOTO 390
380 J=P*.8
390 H=H+(P*Q)
400 K=K+(J*Q)
410 RETURN
```

```
10 !    " THIS IS A MULTIPLICATION LEARNING PROGRAM "
20 !
30 !    " FOLLOW THE INSTRUCTIONS BY THE COMPUTER "
40!
50 INPUT "  PUT IN THE TIMES TABLE VALUE YOU WANT    ", A
60 LET B = A*6
70 LET C=A*2
80 LET D=A*9
90 LET E = A*5
100 LET F = A*11
110 LET G = A*4
120 LET H = A*7
130 LET J = A*12
140 LET K = A*3
150 LET L = A*8
160 LET M = A*10
170 !    A,  "   X  6  =  ", \  INPUT " ", P
180 IF P=B THEN 250
190!
200 IF P<>B THEN 210
210!
220 ! "         YOUR ANSWSER IS INCORRECT --  TRY AGAIN       "
230 !
240 GOTO 170
250 IF P = B THEN GOSUB 1020
260 !
270 !   A,  "   X  2  =  ",  \ INPUT " ", Q
280 IF Q=C THEN 340
290 IF Q<>C THEN 300
300 !
310 ! "            YOUR ANSWSER IS INCORRECT -- TRY AGAIN     "
320 GOTO 260
330
340 GOSUB 1020
350 !  A , "   X    9    =  ", \ INPUT " ", R
360 IF R = D THEN 410
370 IF R<>D THEN 380
380 !
390 !   "      YOUR ANSWER IS INCORRECT -- THINK AND TRY AGAIN   "
400 GOTO 350
410 GOSUB 1020
420 !  A,   "   X    5    =  ", \ INPUT " ", S
430 IF S = E THEN 480
440 IF S<>E THEN 450
450 !
460 !  " *******  YOUR ANSWER IS INCORRECT  -  THINK AND TRY AGAIN **"
470 GOTO 420
480 GOSUB 1020
490 !   A,  "   X   11   =  ",\INPUT " ", T
500 IF T=F THEN 550
510 IF T<>F THEN 520
520 !
530 !  "    YOUR ANSWER IS INCORRECT -- THINK HARD--TRY AGAIN  "
540 GOTO 490
550 GOSUB 1020
560 !  A ,  "   X   4   =  ", \ INPUT " ",  U
570 IF U = G THEN 620
580 IF U<>G THEN 590
590 !
```

```
600!  "       YOU ARE WRONG -- YOUR NOT THINKING -- TRY AGAIN   "
610 GOTO 560
620 GOSUB 1020
630 !  A,   "   X    7    =   ",\ INPUT " ", V
640 IF V = H THEN 690
650 IF V<> H THEN 660
660 !
670 ! "      WRONG----WRONG----WRONG---ONCE MORE = WRONG  "
680 GOTO 630
690 GOSUB 1020
700 !  A ,  "    X    12    =   ", \ INPUT " ", W
710 IF W = J THEN 760
720 IF W<> J THEN 730
730 !
740 !  " THE ANSWER YOU PROVIDED IS WRONG -- TRY AGAIN  "
750 GOTO 700
760 GOSUB 1020
770 !  A ,  "    X    3    =   ", \ INPUT " ", X
780 IF X = K THEN 830
790 IF X <> K THEN 800
800 !
810 ! "  YOU MUST KNOW THAT CAN NOT BE THE RIGHT ANSWER-- TRY AGAIN "
820 GOTO 770
830 GOSUB 1020
840 !  A ,  "    X    8    =   ", \ INPUT " ", Y
850 IF Y=L THEN 900
860 IF Y<>L THEN 870
870 !
880 ! "  NOT RIGHT -- THINK IT OVER AND TRY IT AGAIN  "
890 GOTO 840
900 GOSUB 1020
910 !  A,   "    X    10    =   ", \ INPUT " " , Z
920 IF Z = M THEN 970
930 IF Z<>M THEN 940
940 !
950 !  " WRONG ---  JUST ADD A ZERO TO YOUR NUMBER AND TRY AGAIN"
960 GOTO 910
970 GOSUB 1020
980 ! " YOU HAVE COMPLETED THIS SECTION -- DO YOU WANT TO TRY MORE ? "
990 INPUT " YES OR NO  ", W$
1000 IF W$ = "YES" THEN 50
1002 ! " PLEASE -- TRY IT JUST ONE MORE TIME -- CHICKEN  "
1010 END
1020 !
1030 !  "*********  YOU ARE CORRECT    ***********"
1040!
1050 RETURN
1060 END
```

```
10 DIM C$(40)
20!\!". . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . "
21!"TYPE  DESIRED CONVERSION : 3 KG TO LB."
30!
40INPUT "WHAT TO WHAT : ",C$
50 C5=0
60 FOR X = 1 TO 10
70 D$=C$(1,X)
80 IF C$(X,X)=" " THEN EXIT 100
90 NEXT X
100 F=VAL(D$)
110 FOR Y = X+1 TO 40
120 IF C$(Y,Y)=" " THEN EXIT 140
130 NEXT Y
140 F$=C$(X+1,Y-1)
150 FOR W = Y+1 TO 40
160 IF C$(W,W)=" " THEN EXIT 180
170 NEXT W
180 FOR Z = W+1 TO 40
190 IF  C$(Z,Z)="."  THEN EXIT 210
200 NEXT Z
210  U$=C$(W+1,Z-1)
220 IF F$ = "KG"  THEN K=1
230 IF F$ = "GM"  THEN K=2
240 IF F$ = "LB"  THEN K=3
250 IF F$ = "MG"  THEN K=4
260 IF F$ = "OZ"  THEN K=5
270 IF F$ = "MCG" THEN K=6
280 IF F$ = "GAL" THEN K=7
290 IF F$ = "QT"  THEN K=8
300 IF F$ = "PT"  THEN K=9
310 IF F$ = "L"   THEN K=10
320 IF F$ = "CC"  THEN K=11
330 IF F$ = "ML" THEN K=11
340 IF U$ = "KG" THEN J = 1
350 IF U$ = "GM" THEN J = 2
360 IF U$ = "LB" THEN J = 3
370 IF U$ = "MG" THEN J = 4
380 IF U$ = "OZ" THEN J = 5
390 IF U$ = "MCG" THEN J = 6
400 IF U$ = "GAL" THEN J = 7
401 IF U$ = "QT" THEN J = 8
402 IF U$ = "PT" THEN J = 9
403 IF U$ = "L"  THEN J = 10
404 IF U$ = "CC" THEN J = 11
405 IF U$ = "ML" THEN J = 11
410 IF K = 1 AND J = 3 THEN C5 = 2.205
420 IF K = 1 AND J= 4 THEN C5 = 10^3
430 IF K = 1 AND J = 4 THEN C5 = 10^6
440 IF K = 1 AND J = 5 THEN C5 = 35.28
450 IF K = 1 AND J= 6 THEN C5 = 10^9
460 IF K= 2 AND J= 1  THEN C5 = 10^-3
470 IF K= 2 AND J= 3  THEN C5 = 2.205*10^-3
480 IF K= 2 AND J  = 4  THEN C5 = 10^3
490 IF K = 2 AND J = 5  THEN C5 = .03527
500 IF K = 2 AND J = 6 THEN C5 = 10^6
510 IF K = 3 AND J = 1  THEN C5 = .45359
520 IF K = 3 AND J = 2  THEN C5 = 453.5924
```

```
530 IF K = 3 AND J = 4  THEN C5 = 453592.4
540 IF K = 3 AND J = 5  THEN C5 = 16
550 IF K = 3 AND J = 6  THEN C5 = 4.535924*10^8
560 IF K = 4 AND J = 1   THEN C5 = 10^6
570 IF K = 4 AND J = 3   THEN C5 = 2.205*10^-6
580 IF K = 4 AND J = 2   THEN C5 = 10^-3
590 IF K = 4 AND J = 5   THEN C5 = .03527*10^-3
600 IF K = 4 AND J = 6  THEN C5 = 10^3
610 IF K = 5 AND J = 3   THEN C5 = .0625
620 IF K = 5 AND J = 1   THEN C5 = 28.3495*10^-3
630 IF K = 5 AND J  = 2   THEN C5 = 28.3495
640 IF K = 5  AND J = 4   THEN C5 = 28.3495*10^3
650 IF K = 5 AND J = 6  THEN C5 = 28.3495*10^6
660 IF K = 6 AND J = 3  THEN C5 = 2.205*10^-9
670 IF K = 6 AND J = 1  THEN C5 = 10^-9
680 IF K = 6 AND J = 2  THEN C5 = 10^-6
690 IF K  = 6 AND J = 5  THEN C5 = .03527*10^-6
700 IF K = 7 AND J = 8 THEN C5 = 4
710 IF K = 7 AND J = 9 THEN C5 = 8
720 IF K = 7 AND J = 5 THEN C5 = 128
730 IF K = 7 AND J = 10  THEN C5 = 3.785
740 IF K = 7 AND J = 11  THEN C5 = 3785
750 IF K = 8  AND J = 7 THEN C5 = .25
760 IF K = 8  AND J = 9  THEN C5 = 2
770 IF K = 8  AND J = 5  THEN C5 = 32
780 IF K = 8  AND J = 11  THEN C5 = 946.3
790 IF K = 9  AND J = 7 THEN C5 = .125
800 IF K = 9  AND J = 8  THEN C5 = .5
810 IF K = 9  AND J = 5  THEN C5 = 16
820 IF K = 8  AND J = 10   THEN C5 = .4732
830 IF K = 9  AND J = 11  THEN C5 = 473.2
840 IF K = 5  AND J = 7 THEN C5 = 7.8125*10^-3
850 IF K = 5  AND J = 8  THEN C5 = .0625
860 IF K = 5  AND J = 9  THEN C5 = .125
870 IF K = 5  AND J = 10   THEN C5 = .0295
880 IF K = 5  AND J = 11  THEN C5 = 473.2
890 IF K = 10   AND J = 7 THEN C5 = .2642
900 IF K = 10   AND J = 8  THEN C5 = 1.057
910 IF K = 10   AND J = 9  THEN C5 = 2.113
920 IF K = 10   AND J = 5  THEN C5 = 33.818
930 IF K = 10   AND J = 11 THEN C5 = 10^3
940 IF K = 11  AND J = 7 THEN C5= 2.642*10^-4
950 IF K = 11  AND J = 8  THEN C5= 1.0567*10^-3
960 IF K = 11  AND J  = 9  THEN C5= 2.1132*10^-3
970 IF K = 11  AND J = 5  THEN C5= .033818
980 IF K = 11  AND J = 10   THEN C5= 10^-3
990 IF F$ = U$ THEN C5 = 1
995!
1000 IF C5 = 0 THEN !TAB(15),"THE DESIRED CONVERSION IS NOT IN THIS PROGRAM."
1005 GOTO 1030
1010 !
1020 ! TAB(20),F," ",F$," IS EQUAL TO: ",C5*F," ",U$
1030!
1040 GOTO 20
```

```
1 REM  THIS IS A NUMBER GAME.  THE COMPUTER IS THINKING OF A
2 REM  NUMBER BETWEEN 0000 AND 9999.  BASED ON THE CLUES THAT
3 REM  YOU GET, YOU SHOULD BE ABLE TO GUESS THE NUMBER IN SIX TURNS.
10!
20!
30!
40!
50!
60 LET N = 0
70!
80!
90 FOR A = -1 TO 9
100 FOR B = -1 TO 9
110 FOR C = -1 TO 9
120 FOR D = -1 TO 9
130 LET A = INT(RND(A)*10)
140 LET B = INT(RND(B)*10)
150 LET C = INT(RND(C)*10)
160 LET D = INT(RND(D)*10)
170 IF A=B THEN 10
180 IF A = C THEN 10
190 IF A= D THEN 10
200 IF B = D THEN 10
210 IF C = D THEN 10
220 IF B=C THEN 10
230 LET N = N + 1
240 ! " PUT YOUR NUMBER IN SEPERATED BY COMMAS--1,2,3,4 "
250 INPUT A1,B1,C1,D1
260 LET J = 0
270 LET K = 0
280 IF A1 = A THEN J=J+1
290 IF A1 = A THEN K=K+1
300 IF A1 = B THEN K=K+1
310 IF A1 = C THEN K=K+1
320 IF A1 = D THEN K = K+1
330 IF B1 = B THEN J=J+1
340 IF B1 = A THEN K=K+1
350 IF B1 = B THEN K=K+1
360 IF B1 = C THEN K=K+1
370 IF B1 = D THEN K=K+1
380 IF C1 = C THEN J=J+1
390 IF C1 = A THEN K=K+1
400 IF C1 = B THEN K=K+1
410 IF C1 = C THEN K=K+1
420 IF C1 = D THEN K=K+1
430 IF D1 = D THEN J=J+1
440 IF D1 = A THEN K=K+1
450 IF D1 = B THEN K=K+1
460 IF D1 = C THEN K=K+1
470 IF D1 = D THEN K=K+1
480 ! "  YOU HAVE  ", K ,  "  RIGHT NUMBERS "
490 ! "  OF WHICH  ",J, "  IS/ARE IN THE RIGHT POSITION "
500!
```

```
510 IF J = 4 THEN 530
520 GOTO 230
530 ! "************  YOUR ANSWSER IS CORRECT  ************** "
540 !
550 ! "IT TOOK YOU  ",  N ,  " TURNS TO GET IT RIGHT "
560 !  "         YOUR RATING IS :      "
570!
580 IF N<5 THEN 680
590 IF N=5 THEN 700
600 IF N=6 THEN 700
610 IF N= 7 THEN720
620 IF N= 8 THEN720
630 IF N=9THEN720
640 IF N = 10 THEN 740
650 IF N = 11 THEN 740
660 IF N = 12 THEN 740
670 IF N > 12 THEN 760
680 ! "    :::::::::  YOU ARE A GENIUS -- OR YOU ARE VERY LUCKY ::::::"
690 GOTO 780
700 ! "     *******  YOUR NUMERIC LOGIC IS ABOVE AVERAGE  ********"
710 GOTO 780
720 ! "   ~~~~~~~~~~~ YOUR NUMERIC LOGIC IS AVERAGE  ~~~~~~~~~~~~~~"
730 GOTO 780
740 !"      ########## YOUR NUMERIC LOGIC NEEDS HELP  ########"
750 GOTO 780
760!"  XXXXXXXXX  YOUR NOT TOO BRIGHT - OR YOU ARE NOT TRYING  XXXXXX"
770 GOTO 780
780 !
790 !
800!
810 ! "           WOULD YOU LIKE TO TRY AGAIN ???        "
820 INPUT           "  YES OR NO  :  ",T$
830!
840 IF T$ = "YES" THEN 10
850!
860 ! " CHICKEN--CHICKEN--CHICKEN--CHICKEN--CHICKEN--CHICKEN  "
870 NEXT
880 NEXT
890 NEXT
900 NEXT
910 END
```

```
1 REM  THIS IS A GAME LIKE THE CARD GAME "IN-BETWEEN"
2 REM  YOU BET AGAINST THE POT IF THE NEXT NUMBER DISPLAYED
3 REM  IS IN-BETWEEN THE TWO "CARDS" SHOWN.
10 Q2=100
20!"          ------------------------------------  "
30!"          T H E  P O T  H A S    :  ",$$10F2,Q2
40!"          ------------------------------------  "
50 INPUT"  ",A$
60 X=INT(RND(0)*13+1)
70 Y=INT(RND(0)*13+1)
80 IF X=Y THEN 60
90 IF X=Y+1 THEN 60
100 IF X=Y-1 THEN 60
110 K=ABS((((ABS(X-Y)-1)/13)*100)-100)
120 K2=(((K/50)/1)+(K-50)-(2*(100/K)))*3/5.25
130 IF K2 < .5 THEN K2=.5
140 IF ABS(X-Y)-1 = 6 THEN K2=1
150!"..............................................................."
160!"  C A R D   N U M B E R   O N E    ",X,"      RISK  IS  "
170!TAB(45),%5F1,ABS((((ABS(X-Y)-1)/13)*100)-100)," % AGAINST"
180!"  C A R D   N U M B E R   T W O    ",Y,"     ",ABS(X-Y)-1,"  CARDS"
190!
200!" HOUSE PAYS   ",$$10F2,K2,"   FOR EVERY DOLLAR BET "
210!"..............................................................."
220 INPUT "PLEASE PLACE YOUR  B E T  :  $  ",Q
230!" "
240 Z=INT(RND(0)*13+1)
250!"   N E X T  C A R D  U P  I S  A  :  ",Z
260!
270 IF Q=0 THEN 20
280 IF Z = X THEN 330
290 IF Z = Y THEN 330
300 IF X>Y THEN 330
310 IF X<Y THEN 360
320 IF X=Y THEN 240
330 A=X
340 B=Y
350 GOTO 380
360 B=X
370 A=Y
380 IF Z>A THEN410
390 IF Z<B THEN410
400 GOTO 460
410!"              ## Y O U  L O O S E ##   "
420!
430 Q2=Q2+Q
440 Q4=Q4-Q
450 GOTO 510
460!" $$$$$$$$$$$$$ Y O U  W I N  $$$$$$$$$$$$$$$$"
470!
480 Q2=Q2-Q*K2
490 Q4=Q4+Q*K2
500 GOTO 510
510 IF Q4 > 0 THEN 530
520 IF Q4 < 0 THEN 560
530!" Y O U  N O W  H A V E  W O N  :  ",%$10F2,Q4
540 IF Q2 < 1 THEN10
550 GOTO 20
560!" Y O U  N O W  H A V E  L O S T  :  ",%$10F2,Q4
570 GOTO 20
```

```
1 REM  THIS IS AN INVENTORY COUNTING PROGRAM
2 REM  FIRST YOU MUST CREATE A TYPE 3 DATA FILE (ANY NAME)
3 REM  ALL DATA STATEMENTS MUST HAVE FOLLOWING FORMAT
4 REM
5 REM    ITEM #  ,  "ITEM NAME"  ,  ITEM CLASS  ,  ITEM COST
6 REM
7 REM  THE ITEM NAME MUST BE ENCLOSED IN QUOTATION MARKS
30!
40 INPUT " NAME OF PERSON RUNNING PROGRAM ",K$
50INPUT  "INVENTORY DATE           :  ",D$
60!
70 DIM E$(24)
80!
90INPUT "  FILE NAME  :  ", F$
100 OPEN #0,F$
110 IF TYP(0) THEN 150
120 IF TYP(0)=1 THEN 130
130 READ #0,N
140 GOTO 110
150 !"  FILE IS READY FOR INVENTORY UPDATE"
160!
170!
180 READ N,E$,C,O
190 IF N=1 THEN 410
200 ! N,"   ",E$
210 INPUT " NO. OF UNITS IN INVENTORY  =  ", H
220 !
230! "ITEM#    DISCRIPTION      CLASS    UNIT$   ON HAND"
240!
250!  TAB(7),E$,TAB(29),C,TAB(38),O,TAB(48),H
260!"..............................................."
270 WRITE#0,N,E$,C,O,H
280 GOTO 180
290!"+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
300!
310 REM   YOU CAN HAVE AS MANY ITEMS IN DATA STATEMENTS AS
311 REM   YOU HAVE AVAILABLE MEMORY.  24K = 300 ITEMS
312 REM
313 REM    ITEM #    ITEM NAME    ITEM CLASS    ITEM COST
320 DATA 101,"ORANGES",1,.23
330 DATA 102,"GRAPES",1,.07
340 DATA 104, "EGGS",2,.79
350 DATA 105, "MILK",2,1.25
360 DATA 106, "SOUP",3,.21
370 DATA 107,"CORN",3,.35
390DATA 0,"END-TYPE  0 ",0,0
400DATA 1,"0",0,0
410!
420 !"  YOU ARE NOW FINISHED WITH THE INVENTORY INPUT "
430 WRITE #0,D$,B1,K$
440!
450 !"   THANK YOU VERY MUCH  ",K$
460 CLOSE # 0
470 END
```

```
10  REM   THIS INVENTORY ACCOUNTING PROGRAM MUST BE RUN WITH
20  REM   THE DATA FROM THE INVENTORY COUNTING PROGRAM THAT
30  REM   WAS WRITTEN  TO THE INVENTORY FILE CREATED BY THAT
40  REM   PROGRAM
50  REM
60  REM   THIS PROGRAM WILL GIVE ITEM TOTALS AND EXTENSIONS
70  REM   AND  TOTAL DOLLARS FOR 10 SEPERATE CLASS ITEMS
80  REM   AND  TOTAL OF ALL ITEMS IN LIKE CLASS ITEMS
90 REM
100 !" THIS IS THE INVENTORY EXTENDING AND LISTING PROGRAM"
110!"  IT IS RUN WITH THE INVENTORY COUNT DATA DISC "
120 DIM E$(24)
130!
140 INPUT "DATE OF THIS INVENTORY RUN  :  ",D$
150 INPUT "THIS PROGRAM RUN BY  :  ", K$
160 INPUT " INVENTORY FILE NAME ",F$
170 OPEN #0,F$
180!"...................................................................."
190!"ITEM#     DESCRIPTION        CLASS   UNIT$  COUNT      COST"
200!"...................................................................."
210 T=0
220 T1=0
230 T2=0
240 T3=0
250 T4=0
260 T5=0
270 T6=0
280 T7=0
290 T8=0
300 T9=0
310 READ #0,N,E$,C,O,H
320 IF N=0 THEN 680
330 IF H=0 THEN 310
340 IF C=1 THEN 450
350 IF C=2 THEN 470
360 IF C=3 THEN 490
370 IF C=4 THEN 510
380 IF C=5 THEN 530
390 IF C=6 THEN 550
400 IF C=7 THEN 570
410 IF C=8 THEN 590
420 IF C=9 THEN 610
430 IF C=10 THEN 630
440!
450 T=T+O*H
460 GOTO 660
470 T1=T1+O*H
480 GOTO 660
490 T2=T2+O*H
500 GOTO 660
510 T3=T3+O*H
520 GOTO 660
530 T4=T4+O*H
540GOTO660
550 T5=T5+O*H
560GOTO660
570 T6=T6+O*H
580GOTO660
```

```
590 T7=T7+O*H
600GOTO660
610 T8=T8+O*H
620GOTO660
630 T9=T9+O*H
640GOTO660
650 GOTO 660
660!N,TAB(7),E$,TAB(31),C,TAB(36),O,TAB(45),H,TAB(55),%8F2,O*H
670 GOTO 310
680!
690!":::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::"
700!  "FRUIT AND VEGTABLES            = # 1     ",%$C#10F2,T
710!  "DAIRY PRODUCTS                 = # 2     ",T1
720!  "CANNED GOODS                   = # 3     ",T2
730!  "INVENTORY ADJUSTMENT          =  # 4    ",T3
740!  "A GOODS                        = # 5     ",T4
750!  "B GOODS                        = # 6     ",T5
760!  "C GOODS                        = # 7     ",T6
770!  "D GOODS                        = # 8     ",T7
780!  "E GOODS                        = # 9     ",T8
790!  "F GOODS                        = #10     ",T9
800 REM
810 F1=T+T1+T2+T3+U6+U8
820 R1=T4+T5+T6+T7+T8+T9+U+U1+U2+U3+U4+U5+U7+U9+W+W1
830!"*****************************************************************"
840 E1=F1+R1
850!
860!"   TOTAL FINISHED PRODUCT        :  $  ",F1
870!
880!"   TOTAL RAW MATERIALS           :  $  ",R1
890!
900!"   TOTAL ALL INVENTORY ITEMS     :  $  ",E1
910 ?$#
920!"*****************************************************************"
930 CLOSE #0
940 END
```

```
10 ! CHR$(139)
20 FOR N=1 TO 4 \ ! CHR$(19), \ NEXT N
30 READ X,Y
40 IF X=0 THEN 630
50 FILL(52224 + X + 64* Y ) , 160
60 GOTO 30
70 REM LEFT WALL BOTTOM TO TOP
80 DATA 6,15,6,14,6,13,6,12,6,11,6,10,6,9
90 DATA 7,15,7,14,7,13,7,12,7,11,7,10,7,9
100 REM BEAN LEFT TO RIGHT
110 DATA 4,9,5,9,6,9,7,9,8,9,9,9,10,9,11,9,12,9,13,9,14,9,15,9,16,9,17,9
120 DATA 17,9,18,9,19,9,20,9,21,9,22,9,23,9,24,9,25,9,26,9,27,9
130 DATA 28,9,29,9,30,9,31,9,32,9,33,9,34,9,35,9,36,9,37,9,38,9
140 REM RIGHT WALL TOP TO BOTTOM
150 DATA 38,9,38,10,38,11,38,12,38,13,38,14,38,15
160 DATA 37,9,37,10,37,11,37,12,37,13,37,14,37,15
170 REM BEAM EXTENSION
180 DATA 38,9,39,9,40,9,41,9
190 REM LEFT ROOF BOTTOM TO TOP
200 DATA 6,9,7,9,6,8,7,8,7,7,8,7,9,7,9,6,8,6,10,5,9,5
210 DATA 10,4,11,4,11,3,12,3,12,2,13,2,13,1,14,1
220 REM ROOF BEAN LEFT TO RIGHT
230 DATA 14,1,15,1,16,1,17,1,18,1,19,1,20,1,21,1,22,1,23,1
240 DATA 24,1,25,1,26,1,27,1,28,1,29,1,30,1,31,1,32,1,33,1
250 REM RIGHT ROOF TOP TO BOTTOM
260 DATA 32,1,33,1,33,2,34,2,34,3,35,3,35,4,36,4,36,5,37,5,37,4
270 DATA 37,6,38,6,38,7,39,7,39,8,40,8,40,9,41,9,41,9
280 REM FLOOR LEFT TO RIGHT
290 DATA 4,15,5,15,6,15,7,15,8,15,9,15,10,15,11,15,12,15,13,15
300 DATA 14,15,21,15,22,15,23,15,23,15,24,15,25,15,26,15,27,15
310 DATA 28,15,29,15,30,15,31,15,22,15,32,15,33,15,34,15,35,15
320 DATA 35,15,36,15,37,15,38,15,39,15
330 REM DOOR \ BOTTOM LEFT TO TOOP TO RIGHT TO BOTTOM RIGHT
340 DATA 14,14,21,15,14,13,14,12,14,11,15,11,16,11,17,11,18,11
350 DATA 19,11,20,11,21,11,21,12,21,13,21,14,21,15
360 REM WINDOW
370 DATA 27,13 ,28,13,29,13,30,13,31,13,32,13,33,13
380 DATA 33,12,33,11,32,11,31,11,30,11,29,11,28,11
390 DATA 28,12, 34,13
400 REM GRACE
410 DATA 38,9,40,9,41,9,42,9,43,9,44,9,45.9,46,9,47,9,48,9,49,9
420 DATA 50,9,51,9,52,9,53,9,54,9,55,9,56,9,57,9,58,9,60,9
430 DATA 59,9,59,8,59,7,59,6,59,5,59,4,59,10,59,11,59,12,59,13
440 DATA 59,14,59,15
450 REM FLAG
460 DATA 59,4,58,4,57,4,56,4,55,4,54,4,52,4,51,4
470 DATA 59,6,58,6,57,6,55,6,54,6,53,6,51,6
480 REM AUTO
490 DATA 42,15,55,15,43,14,44,14,45,14,46,14,47,14,48,14,49,14
500 DATA 42,14,41,14,50,14,51,14,52,14,53,14,54,14,55,14,56,14
510 DATA 43,13,44,13,45,13,46,13,47,13,48,13,49,13,50,13,51,13
520 DATA 52,13,53,13,54,13
530 DATA 44,12,53,12,49,12
540 DATA 43,11,44,11,45,11,46,11,47,11,48,11,49,11,50,11
550 DATA 51,11,52,11,53,11,54,11,55,11
560 DATA 10,6,10,5,10,4,10,3,10,2,10,1,9,1,9,2,9,3,9,4,9,5,9,6
570 DATA 8,1,8,2,8,3,8,4,8,5,8,6
580 REM UP WINDOW
590 DATA 19,4,20,4,21,4,22,4,23,4,24,4,25,4,26,4,27,4
600 DATA 19,5,19,6,19,7,20,7,21,7,22,7,23,7,24,7,25,7,26,7,27,7
610 DATA 27,6,27,5,23,5,23,6
620 DATA 0,0
630 RESTORE
640 FOR N= 1 TO 1000\ NEXT N
650 GOTO 10
660 END
```

```
1 REM  THIS PROGRAM PREPARES THE EMPLOYEE DATA FILE (TYPE 3)
2 REM  TO BE USED WITH THE PAYROLL PROGRAM.  ALL OF THE
3 REM  QUESTIONS MUST BE ANSWERED.  AN HOURLY RATE MUST BE USED
4 REM  EVEN FOR SALARIED EMPLOYEES- BASED ON 40 HOURS PER WEEK
5 REM  HOURLY EMPLOYEES MUST HAVE AN EMPLOYEE NUMBER GREATER
6 REM  THAN 200 --- SALARIED EMPLOYEES A NUMBER LESS THAN 200.
10 DIM N$(30),S$(11)
20 OPEN #0, "EMPLOYEE"
30 INPUT " EMPLOYEE NUMBER         :  ", N
40 IF N = 0 THEN 220
50 INPUT " EMPLOYEE NAME           :  ",N$
60 INPUT " SOCIAL SECURITY NO.     :  ",S$
70 !" TYPE IN FOR MARITAL STATUS : "
80 INPUT " 1 - SINGLE * 2 - MARRIED: ",M
90 INPUT " EMPLOYEE'S PAY RATE     :  ",R
100 INPUT " TOTAL NO. DEPENDENTS   :  ",D
110 !" TAKING DEPENDENT INSURANCE ? "
120 INPUT " 1 FOR YES   2 FOR NO  :  ",I
130 !
140 IF TYP(0) = 0 THEN 200
150 IF TYP(0)=2 THEN 160
160 READ #0, N1,N1$,S1$,M1,R1,D1,I1
170!" THE LAST NUMBER READ ON THE FILE WAS :  ",N1
180!" THE LAST NUMBER WRITTEN ON THE FILE  :  ",N
190 GOTO 140
200 WRITE #0, N,N$,S$,M,R,D,I
210 GOTO 30
220 CLOSE #0
230 END
240!
250!
260!
270 OPEN #0,"EMPLOYEE"
280 IF TYP(0)=0 THEN 320
290 IF TYP(0)=2 THEN 300
300 READ #0,N1,N1$,S1$,M1,R1,D1,I1
310 GOTO280
320 !" THE LAST EMPLOYEE NUMBER ON FILE IS : ",N1
330 CLOSE#0
340 END
```

```
1 REM  THIS IS A PAYROLL PROGRAM WHICH CALCULATES WITHOLDING
2 REM  AND FICA, ALLOWS FOR OTHER DEDUCTIONS.  IT REQUIRES
3 REM  A TYPE 3 EMPLOYEE-DATA FILE (INCLUDED IN THIS SECTION)
4 REM  IT IS BASED ON A WEEKLY PAYROLL AND HANDLES BOTH
5 REM  SALARIED (NOS. 1 TO 199) AND HOURLY (NOS. > 200)
6 REM  THE WITHHOLDING TAX IS BASED ON 1978 RATES
7 REM  TO UPDATE CHANGE LINES 430 TO 600
10  DIM N$(30),S$(11)
20 K4=0
30 INPUT " THIS PAYROLL IS FOR THE PERIOD ENDING : ", D$
40!\!"-----------------------------------------------------"
50 OPEN #0, "EMPLOYEE"
60 INPUT "WHAT IS THE EMPLOYEE NUMBER :", N9
70 I7=0\H2=0\F8=0
80 IF N9 = 0 THEN 1000
90 IF N9<200 THEN K4 = K4 +1
100 READ #0, N,N$,S$,M,R,D,I
110 IF N = N9 THEN 160
120 IF TYP(0)=2 THEN 100
130 IF TYP(0)=0 THEN ! "THERE IS NO SUCH NUMBER "
140 CLOSE #0
150 GOTO 50
160!
170!" THE NUMBER  : ",N9," IS FOR : ",N$
180 !
190 IF N9<200 THEN 210
200 GOTO 220
210 H=40\GOTO 260
220 INPUT "TOTAL NO. OF REGULAR HOURS WORKED   : ",H
230 IF  H < 41 THEN 250
240  !"*** NO MORE THAN 40 ***"\GOTO 220
250 INPUT "TOTAL NO. OF OVER-TIME HOURS WORKED : ",H2
260!
270 IF N9>200 THEN 300
280 INPUT "IF COMMISIONS HOW MUCH  :  ",F8
290 F9=F9+F8
300 W = H*R
310 W1 =H2*R*1.5
320 G = W + W1 + F8
330 H5=H5+H
340 H6=H6+H2
350 W6=W6+W1
360 INPUT "AMOUNT OF ADVANCEMENT REPAYMENT : ",D7
370!
380 D1 = D*14.4
390 N1 = G-D1
400 F=G*.0605
410 ON M GOTO 420,520
420!
430 IF N1<33 THEN T1=0
440 IF N1>33 THEN T1=(N1-33)*.16
450 IF N1>76 THEN T1=6.88+(N1-76)*.18
460 IF N1>143THENT1=18.94 +(N1-143)*.22
470 IF N1 >182 THEN  T1=27.52+(N1-182)*.24
480 IF N1>220 THEN T1=36.64+(N1-220)*.28
490 IF N1>297 THEN T1=58.20+(N1-297)*.32
500 IF N1>355 THEN T1=76.76+(N1-355)*.36
510 GOTO 610
520!
530 IF N1<61 THEN T1=0
540 IF N1>61 THEN T1= (N1-61)*.15
```

```
550 IF N1>105 THEN T1 =6.6+(N1-105)*.18
560 IF N1 > 223 THEN T1=27.84+(N1-223)*.22
570 IF N1>278 THEN T1 = 39.94+(N1-278)*.25
580 IF N1>355 THEN T1=59.19+(N1-355)*.28
590 IF N1>432 THENT1=80.75+(N1-432)*.32
600 IF N1>509 THEN T1=105.39+(N1-509)*.36
610!
620!
630 !"EMPLOYEE NUMBER            :  ",TAB(35),$10F0,N
640 !"                              ",TAB(35),N$
650 !"SOCIAL SECURITY #          :  ",TAB(35),S$
660 IF N9<200 THEN 690
670 !"PAY RATE                   :  ",TAB(35),$$10F2,R
680 GOTO 710
690 R2=R*40
700 !"PAY RATE                   :  ",TAB(35),$$10F2,R2
710 !"TOTAL DEPENDENTS           :  ",TAB(35),$10F0,D
720 !"PAY PERIOD ENDING          :  ",TAB(38),D$
730 !"GROSS PAY                  :  ",TAB(35),$10F2,G
740 IF N9<200 THEN 780
750!H," REGULAR HOURS =         : ",TAB(35),$10F2,H*R
760 IF H2=0 THEN 780
770!H2," OVER-TIME   =            : ",TAB(35),$10F2,H2*R*1.5
780 IF F8>1 THEN 800
790 GOTO 810
800 ! "COMMISIONS                 : ", TAB(35),$10F2,F8
810 ! "FICA WITHELD   =          : ",TAB(35),$10F2,F
820 ! "WITHOLDING TAX =          : ",TAB(35),$10F2,T1
830 IF I = 2 THEN 880
840 I2 = 13.12\I3=I3+I2
850 ! "DEPENDENT INSURANCE        : ",TAB(35),$10F2,I2
860 I7=13.12
870 IF D7=0 THEN 900
880 ! "ADVANCE DEDUCTION          : ",TAB(35), $10F2,D7
890 D8=D8+D7
900 !
910!"*** NET PAY  : ",$$C10F2,G-F-T1-I7-D7
920!
930 N2=G-F-T1-I7-D7
940 G1=G1+G
950 F1=F1+F
960 T2=T2+T1
970 N3=N3+N2
980 CLOSE #0
990 GOTO 50
1000!\!\!"*****************************************************"
1010!\!\!
1020 ! " TOTALS FOR PAYROLL PERIOD ENDING  :  ", D$
1030!
1040 !" TOTAL GROSS PAYROLL        :  ",$#$C10F2, G1
1050 !" TOTAL FICA                 :  ",F1
1060 !" TOTAL WITHHOLDING TAX      :  ",T2
1070 !" TOTAL DEPENDENT INSURANCE  :  ",I3
1080 !" TOTAL ADVANCE DEDUCTIONS   :  ",D8
1090 !" TOTAL NET PAYROLL          :  ",N3
1100!
1110 IF N3<>G1-F1-T2-I3-D8 THEN !"ERROR IN NET PAYROLL ********"
1120!$#
1130 !" TOTAL REGULAR HOURS PAID  :  ",H5-(K4*40)
1140 !" TOTAL OVER-TIME HOURS PAID:  ",H6
1150 !" TOTAL OVER-TIME WAGES PAID:  ",$$C10F2,W6
1160 !" TOTAL COMMISIONS PAID     :  ",$$C10F2,F9
```