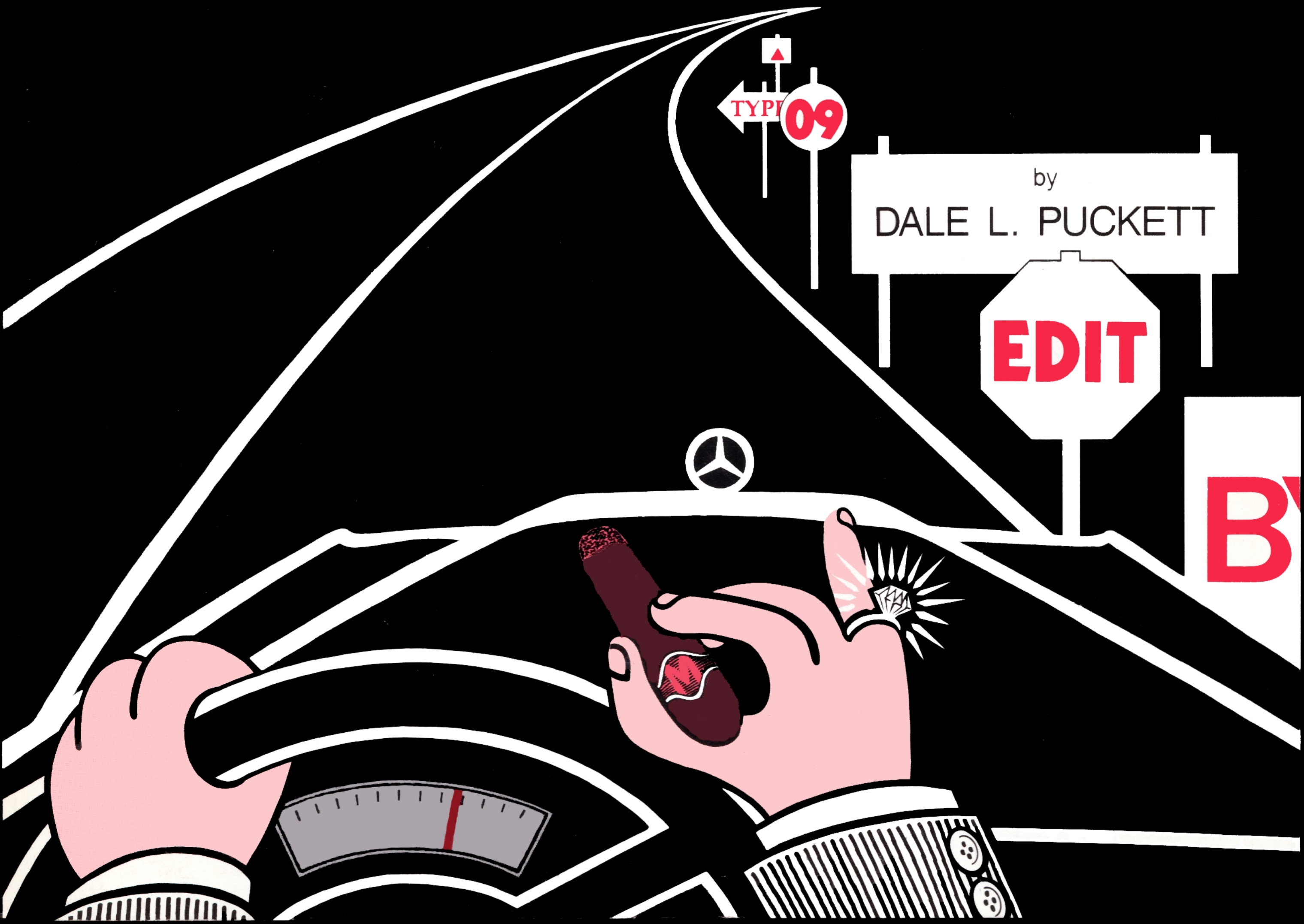
Radio Shack®

Cat. No. 26-3189

The Official BASIC09 Tour Guide

Your Introduction to BASIC09, a State-Of-The-Art Programming Language from MICROWARE



by

DALE L. PUCKETT

The Official BASIC09 Tour Guide

Your Introduction to BASIC09, a State-Of-The-Art Programming Language from MICROWARE

by DALE L. PUCKETT

MICROWARE SYSTEMS CORPORATION
1866 N.W. 114th STREET / DES MOINES, IOWA 50322

Second edition. Publication date: November 1, 1985

ISBN 0-918035-00-7

1 2 3 4 5 6 7 8 9 10

Book design and illustrations by: George T. Edwards (The Art Board II)
Typesetting by: Type-O-Graphics Two, Inc.

table of contents

the philosophy

Got your attention, didn't I? Why would anyone that hates **BASIC** write a book about something called "BASIC09"?

The answer to that question is simple. BASIC09 is much more than **BASIC.** It is a complete programming system that gives you the friendly interactive feel of **BASIC** while delivering the modularity, structure and readability of the **PASCAL** language. We'll be showing you the advantages of these characteristics throughout the book. But for now, let's get back to that first sentence.

Forget that you love **BASIC** and hate **PASCAL.** Or, vice versa. Just look at the mechanics of the sentence.

It is a simple sentence. It has only one subject, one verb, and one object. It is inherently clear.

As simplicity is the key to success for the writer, it is also the key to success for the programmer. I hope it will be the key to the success of this book.

USING THIS BOOK

The Official BASIC09 Tour Guide has four parts. The first part teaches you the mechanics, the second introduces you to the language itself, the third is a complete reference guide that shows each OS-9 operating system command, and finally, the fourth lists each BASIC09 keyword in alphabetical order.

Part One introduces you to the operating system. You'll learn how to use the keyboard and editor to enter your own programs. Then, you'll find out how to run your programs using BASIC09's system commands and learn to debug them with the debugger.

Part Two teaches you the language itself. You'll find that writing good programs in BASIC09 is easy and quite natural.

The two reference guides presented will make life easy after you learn the basics. You'll find illustration, short procedures and sample runs to project the concept of OS-9 commands and BASIC09 keywords.

CHAPTER DESCRIPTIONS

Chapter One is for fun. It gives you a chance to run a short program before you dive into the book.

Chapter Two gets down to business. You'll be introduced to several special keyboard functions as we show you how to talk to your computer.

Chapter Three gives you a tour of Microware's OS-9 operating system. If you've never been able to master chewing gum and juggling tennis balls at the same time — hold on. We'll reveal the magic of multi-tasking.

Chapter Four covers BASIC09's editor. You'll be entering your own programs in no time.

Chapter Five shows you how to run your programs. You'll also learn about **LOAD**ing, **LIST**ing, and **KILL**ing them.

Chapter Six deals with the inevitable. We'll show you how to find those pesty parasites that keep sneaking into your programs.

Chapter Seven will make you feel like a motion picture director after you learn to **TYPE** variables. You'll discover the difference between **INTEGER** and **REAL** numbers and be introduced to **BYTE, STRING,** and **BOOLEAN.** Will **TRUE** ever be **FALSE?**

Chapter Eight shows you how to express yourself clearly when writing BASIC09 programs. You'll be introduced to statements, functions, and many operators.

Chapter Nine talks about structure. It will put you in control of your computer. No longer will your programs be GOing TO visit strange line numbers.

Chapter 10 shows how your programs talk to the outside world. You'll learn about input and output paths and find out how you can keep permanent files.

Chapter 11 is for the purist who knows in his heart that form follows function. You'll learn everything you ever wanted to know about **PRINT USING.**

Chapter 12 shows you how to make BASIC09 procedures **RUN** other procedures. You'll see why it's more fun to turn a big job into a number of small jobs after we introduce you to modularity. Then, we'll give you a special treat and show how recursion can help you solve complex programming problems.

Chapter 13 shows you how to make BASIC09 run 6809 machine language subroutines. You probably won't need this information though — almost every function you can dream up is already in BASIC09.

Chapter 14 gives you some fairly good sized Basic09 programs to study and to use.

HOW THIS BOOK WAS CREATED

The manuscript of this book was prepared by the author and the editors using word processing and spelling checker programs running on a 6809 microcomputer under the OS-9 Level II operating system.

The manuscript was transmitted directly from an OS-9 computer to a Compugraphic typesetting system. This process helped us ensure that the accuracy of the contents of this book, especially of the sample programs.

ACKNOWLEDGEMENTS

Several people deserve special credit. Bill Ball and Paul Powers, fellow Coast Guard photojournalists and neophyte computer programmers, read every page of the manuscript to insure that it could be understood by beginners. Ball also helped compile the excellent encyclopedic listing of BASIC09 keywords in Part IV.

Thanks go to Tom Westhoff, an instructor at Willmar Area Vocational-Technical Institute in Minnesota. His advice and editorial assistance helped make this a better book for beginning programmers.

We all owe a great deal of thanks to Ken Kaplan, Larry Crane, and Bob Doggett at Microware and Terry Ritter (formerly at Motorola) for designing this state-of-the-art language.

And, very special thanks go to my wife Esther who put up with the long hours and creative pressures of this writing project.

CHAPTER ONE

the system if you can't wait

PART ONE

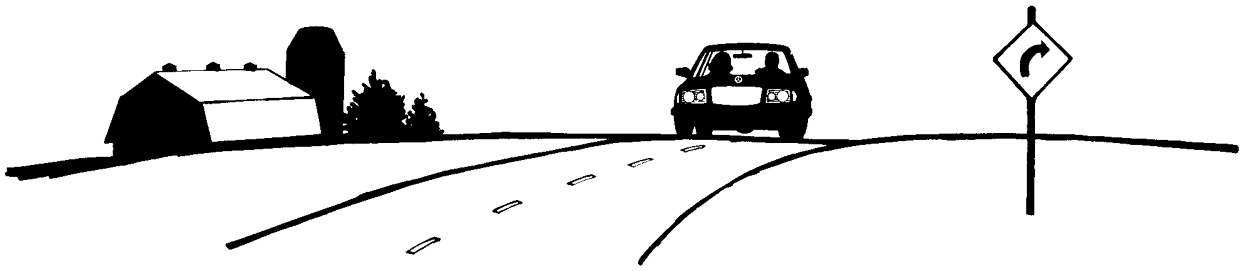
Welcome! You are beginning a friendly, guided tour of a revolutionary new computer language — Microware's BASIC09.

This Chapter is dedicated to those of you who yearn for adventure — people like me, who enjoy playing with a new toy as soon as the package is open.

We'll give you a short procedure to enter and run. After you've run it a few times and the new has worn off, we'll meet you in Chapter Two where our guided tour officially begins. There you'll meet the OS-9 operating system and several special keys on your terminal.

For now, let's play!

Follow these steps and you'll be running the demonstration procedure in no time at all.



a. Turn on your computer and bring the OS-9 operating system to life. If you don't know how, have a friend show you or consult your OS-9 User's Manual.

b. When the system is ready, you should see this prompt:

OS9:

c. Now, type **BASIC09** and then hit the (**RETURN**) key. After a few seconds, you're screen should look like this.

OS9: BASIC09
BASIC09
Copyright 1981 Microware and Motorola
Reproduced Under License
B:

d. Now, type: **EDIT DEMO** and hit the (**RETURN**) key to put you into BASIC09's edit mode: You should see this on the screen:

B: EDIT DEMO
PROCEDURE DEMO
*****
E:

e. The **E:** is a prompt for BASIC09's editor. It means that BASIC09 is ready for you to enter the demonstration procedure.

f. To enter a line, you must type the (**SPACE**) as the first character in the line. After you have hit the spacebar, just type the line as it is printed here, and then hit the (**RETURN**) key.

g. There are several things you should be aware of as you type. First, the (**SPACE**) will appear as a blank character position on your screen. The (**RETURN**) key will also be invisible to you, but BASIC09 will echo it to the screen and cause the star, "*" prompt to be printed on the next line.

```
B: EDIT DEMO (RETURN)
PROCEDURE DEMO
*
E: (SPACE) print (RETURN)
*
E: (SPACE) input "Please type your name here:", name (RETURN)
*
E:
```

h. Now, go ahead and type the rest of the procedure listed below.

i. After you have typed the entire procedure, answer the editor's prompt by typing the letter **Q** to return to BASIC09's system mode. Be sure that you type the **Q** as the first character in the line. If you put a space in front of it, BASIC09 will think you are typing a word that it doesn't know and you'll find out about error messages before we can explain them in this book.

j. As soon as you see the **B:** prompt again, you may run the procedure by typing:

```
B: RUN DEMO (RETURN)
```

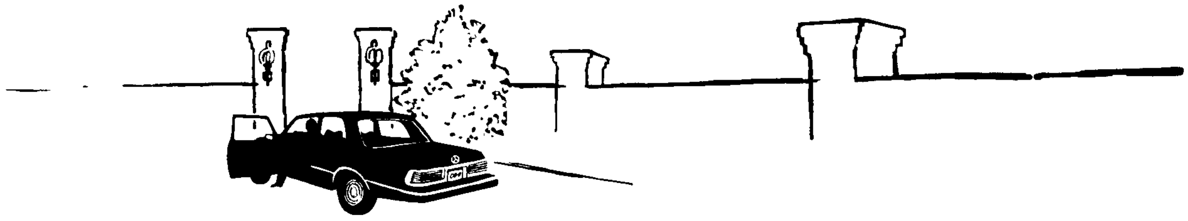
k. Enjoy your first procedure for awhile, and we'll see you soon in Chapter Two.

EXAMPLE NO. 1: DEMO

```
PROCEDURE demo
DIM counter1,counter2:INTEGER
DIM name:STRING[20]

    PRINT
    INPUT "Please type your name here:", name
    PRINT "Hello"; name;"." "Welcome to BASIC09!"
    PRINT
    PRINT "I can help you do many things."
    PRINT "For example, I can print the multiplication"
    PRINT "table for you.  Watch!"
    PRINT
    FOR counter1:=1 TO 9
    FOR counter2:=1 TO 9
    PRINT USING "I4>"; counter1*counter2;
    NEXT counter2
    PRINT
    NEXT counter1
    PRINT
    PRINT "Now that you know I work, please relax"
    PRINT "and turn to Chapter Two.  I hope you'll"
    PRINT "enjoy your guided tour"; name; "."
    PRINT
    END
```

CHAPTER TWO

where do i start?



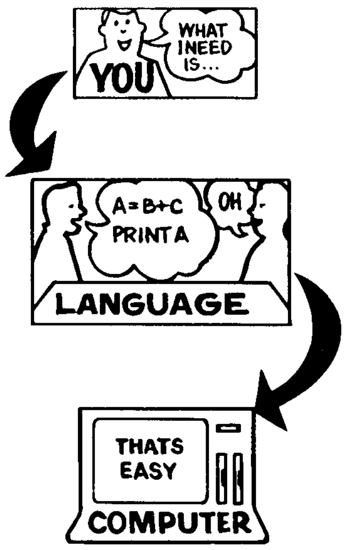
WHERE DO I START? Welcome aboard! Fasten your seatbelt, grab your terminal and hold on! You're about to drive the Mercedes of computer languages — BASIC09.

In this chapter we'll be showing you the magic of the keyboard as you learn how to talk to BASIC09 from your terminal. We'll even throw in a few tricks that will make your life as a programmer easier.

Writing a computer program is similar to writing a novel. The programmer, like the author, must have full command of the language. Yet, the language must do its part.

HOW LANGUAGES WORK: COMPILERS AND INTERPRETERS

Computers, and particularly microcomputers, are extremely simple minded machines. Basically, they work by performing long sequences of very simple individual steps. The secret of their magic is that they can do calculations very quickly (on the order of a million steps per second). But they don't have a prayer of even coming close to understanding human languages. What they do understand are long lists of commands stored in memory as numerical codes. These codes are called "machine language". Unfortunately, raw machine language is as alien to humans as English is to computers.



Programming languages span this communications gap between computers and people. Programming languages are themselves programs that translate special languages to machine language the computer can deal with. The languages are compromises that are concise and simple enough for computers to figure out yet powerful enough for humans to comfortably use to solve complicated problems.

There are two basic ways a programming language can carry out its translation task. One class of languages, called "compilers", take a previously prepared program (called the "source" program) and translate the whole works to a corresponding machine language program (called the "object program") — all in one shot.

The other class of languages are called "interpreters". They let you type in a program which is stored in memory almost exactly as you have typed it in i.e., in "source" form. When you tell the interpreter to run your program, it gets each program statement, one-by-one, figures out the corresponding machine language instructions required, and executes them. The problem is that after it's figured out what a statement means, it has no way to remember the interpretation if the same statement happens to be encountered again later, and this is very often the case. Because the interpretation process also consumes a considerable amount of time, the program runs much slower than the one-shot compiler method. Another disadvantage of interpreter languages is that for a given program, the source version needs more memory storage space than the object version.

Despite the relative slowness and voracious memory appetite of interpreters, they are far and away most widely used on microcomputer systems because interpreters allow changes to be made quickly and conveniently. They also usually provide easier ways to test errant programs. As you will come to appreciate, as you acquire programming experience, these factors can easily tip the scales from the better efficiency of compiler languages to the ease of use of interpreter languages.

INTERACTIVITY

Let us digress. I remember my early years in journalism well. The adage that good stories aren't written, but rather re-written, was alive and well. It was an age when we struggled without word processors.

When I started a story, I:

a. put a piece of paper in the typewriter,
b. typed a lead sentence,
c. read the sentence,
d. tore the paper from the typewriter, and finally,
e. crushed the paper into a little round ball and threw it in the wastebasket.

I repeated those steps — usually in the same order — until:

a. I was satisfied with the lead sentence,
b. had passed my deadline, or
c. there were no more trees in the forest.

The steps I went through to write the story above are similar to the steps you must follow today when using most high level language compilers.

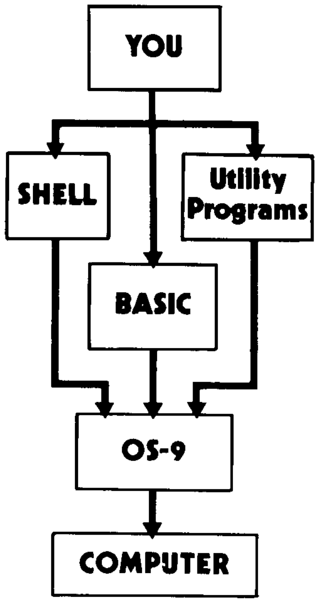
Before you can run a program, you must:

a. type in your program source code using a text editor,
b. save your code to a disk file,
c. run a compiler that produces an object code file on a floppy disk, then
d. load the object file into memory and run it.

What a hassle! And just think, if you make a mistake anywhere along the line, you must start the entire process over again.

Enter our hero, BASIC09 — a complete programming system designed to make your life easy! BASIC09 contains a powerful text editor, a multi-pass compiler, and a run-time debugging package that is entered automatically when an error occurs. These tools are in memory at the same time and can be run instantly by typing a single key. Program development time is shortened. And what does the preceding triple mouthful actually mean? Simply this: BASIC09 IS ACTUALLY A COMPILER-TYPE LANGAUGE CAREFULLY DISGUISED AS AN INTERPRETER SYSTEM. Yes, with BASIC09 you have the best of both worlds—the efficiency of a compiler with the convenience and friendliness of an interpreter!

Throughout the rest of the book we'll be giving you short examples. Type them in and watch them run. Better yet, experiment with them. Make small changes and watch your terminal's screen to see what happens.



The programs in the first several chapters are included to give you something to type in and run while you are learning the mechanics. If you have questions about any BASIC09 keyword, you may look it up in the encyclopedic listing in Part Four of this book.

BASIC09 is a programming language. It runs in an environment called OS-9. OS-9 is an "operating system" that lets BASIC09 and other programming languages talk to the real world. The role of an operating system is to manage the basic operation of the computer, much like a traffic cop. For example, it handles all the complicated input and output for your terminal and disks, etc. If the computer did not have an operating system, every program would have to include its own complicated input/output functions. The operating system always lives in memory along with any other programs being used. The operating system also eliminates possible mayhem by setting standards so data files used by different programs and languages are compatible with each other.

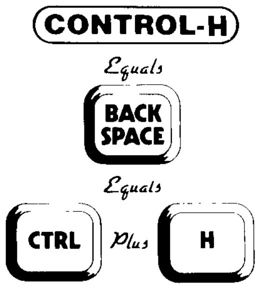
OS-9 connects BASIC09 to the keyboard you type on. It writes letters and numbers on your terminal's screen so you can read them. It prints data on your printer so you can have a hardcopy. It saves programs on floppy disks for you — so you won't have to type them over every time you want to run them. It even lets you run two or more programs at the same time — a process called multi-tasking.

OS-9 has its own library of programs that you can call in to do your dirty work. These programs are called utility programs and we'll show you how to use them in Chapter Three. But first, let's demonstrate the magic of your keyboard.

SPECIAL KEYS

Your keyboard has several special keys which make life easier. They let you correct mistakes, repeat actions or even stop a program in midstream.

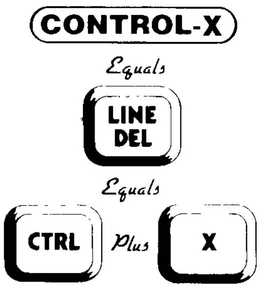
Let's start with the "mistakes". What happens if you mean to type **PRINT** but **PRING** comes out of your fickle fingers. No problem —if you correct it. And if you don't, BASIC09 will let you know.



There are two ways to correct an error. You can hit the backspace key to back the cursor up to the bad character and then type over it. Or, you can hit the line delete key to get rid of the whole mess and start over. It's your choice.

On most terminals the keyboard has a key marked, **BACK SPACE** or **BKSP**. It causes the cursor to back up one character position.

If your terminal does not have this key, you can back the cursor up one position by striking the **H** key while holding down the key marked **CTRL**. Other terminals have a group of four arrow keys that let you move the cursor. On these terminals, the arrow that points to the left can be used.



You say you don't have a key marked, **LINE DELETE.** No problem! Most terminals don't. On OS-9 systems you may delete the line you are typing by striking the **X** while holding down the key marked **CTRL**.

Other special functions let you repeat the previous input line, interrupt a program, redisplay the present input line, exit a program, and wait. The "wait" function gives you a way to stop your terminal from scrolling so you may study a line while listing a long procedure.

THE REPEAT KEY

You'll love the repeat function because of the wear and tear it saves your finger tips. To use it, you hold down the **CTRL** key while typing the letter **A**. You'll find this function really handy when you need to run the same procedure many times.

With BASIC09 and OS-9, there's no need to type a command line over and over. Just type (**CONTROL-A**), and the line will magically reappear. Then, type (**RETURN**) to run the command again.

Try it the next time you turn on your OS-9 computer. Type: **dir** (**RETURN**). Your computer will promptly list the contents of the current data directory.

Then, type: (**CONTROL-A**) (**RETURN**) . Your trusty OS-9 machine will list the directory again. If you think the repeat key is neat now, wait till you use it with a long command line.

Here's another trick you can use after you have typed and saved a few BASIC09 procedures. Suppose you need to list the procedures APROCEDURE, BPROCEDURE ... FPROCEDURE. You can use the repeat key as a short cut. Type:

LIST APROCEDURE (RETURN).

BASIC09 should do what it's told.

On with the magic. Now, Type:

(**CONTROL-A**).

You'll see **LIST APROCEDURE** reappear. Now, hit the **BACK SPACE** key until the cursor is over the **A** in **APROCEDURE.** If your keyboard will auto-repeat, just hold down this key until you arrive.

After you have done this, type a **B** followed by another (**CONTROL-A**), followed by (**RETURN**) . You should see BASIC09 LIST **BPROCEDURE.**

Doesn't that beat typing. Exercise this special key every time you get the chance. You'll save hours.

OTHER SPECIAL KEYS

If you ever need to temporarily stop the execution of a procedure in the middle, you can use the Interrupt Key. On most OS-9 systems, you strike the letter **C** while holding down the **CTRL** key.

You may redisplay the present input line by typing (**CONTROL-D**) or, stop a program by typing (**CONTROL-Q**) . The **Q** stands for quit.

Imagine that you are running a program that prints a long list of numbers on your terminal. The numbers are coming at you so fast that they scroll off the screen before you can read them. What do you do?

Sounds like a good time to try OS-9's special "wait" key. Strike the **W** key while holding down the key marked **CTRL.** The printing should stop. After you have studied the numbers, you may continue printing by hitting any other key.. Try it.

The last special key is marked **ESC** — for **ESCAPE** — on most keyboards. It sends an end-of-file character, to BASIC09 and gives you a way to send an end-of-file signal to procedures that receive data from the keyboard.

There's only one catch. When you send this **ESCAPE** code to BASIC09, you must be sure that you type (**ESCAPE**) as the first character on the line.

OTHER BASIC09 MAGIC

Keep your seatbelt buckled, we haven't told you everything. Are you impatient? Do you often know what you want to do next but find yourself waiting for the computer to finish one task before you can tell it to do another?

Rest easy. Most BASIC09 systems let you "type ahead." This term is a fancy way of saying that while the computer is running one program, you can go ahead and type another command line, or answer the next question if you know what it will be.

In fact, you may stay several command lines in front of the computer. It will execute them one at a time, just as fast as it can. The only catch is that you will be typing blind, which is only a minor slow down however and is much better than sitting around chewing your fingernails.

Keep the faith. In the next chapter, we'll show you how to tell the operating system to go do one job and come right back and ask for another. You'll even learn how to make it do three or four things at the same time.

UP OR DOWN?

Should you type your programs using uppercase letters? Or, would lowercase letters look better?

BASIC programs look nice and are easy to understand when BASIC commands and statements are typed with uppercase letters and variable names are typed with lowercase letters. With many languages, this scheme is a major hassle. With BASIC09, it's a snap.

Here's how you do it. When you type a procedure, leave your keyboard in the lowercase mode and only use the shift key when you want your procedure to print a capital letter. BASIC09's editor will do the rest. When you list the program you'll be in for a pleasant surprise.

In fact, you'll find that BASIC09 does more than capitalize commands. It also automatically indents listings to make procedures easier to read and help you find certain logical errors.

Here's an example:

If you type:

```
for count := first to last
print count
next count
```

BASIC09 will put this into memory:

```
FOR count := first TO last
    PRINT count
NEXT count
```

And speaking of surprises. You'll really do a double take when you make your first mistake while typing a BASIC09 program. But, that's a story for Chapter Four.

SUMMARY

In this chapter you have learned about several special keys that will make life easier. You should know how to:

a. Backspace the cursor
b. Delete a line
c. Repeat a command line
d. Redisplay an input line
e. Interrupt a program
f. Stop the screen from scrolling
g. Stop a program
h. Send an ESCAPE code
i. Type ahead of the computer
j. Type your programs in lowercase letters

Can you remember which control keys are used for a. to h. above? If not, take time out for a review.

It's time to get this buggy headed down the road. Turn on your OS-9 based computer, turn to the next chapter, and you'll learn about one of the most powerful operating systems on a microcomputer today.

CHAPTER THREE

touring Microware's OS-9 operating system

We gave you the first hint of OS-9's power in the last chapter when we introduced you to several special keys on your terminal. If you were impressed then, tighten your seatbelt — "you ain't seen nothing yet!"

As you may have guessed, much of BASIC09's power comes from its environment — Microware's OS-9 operating system.

In this Chapter we'll give you a brief overview of OS-9 and show how BASIC09 procedures can run many operating system utility programs automatically. Of course, you can also run them manually when BASIC09 is in the system mode.

You'll be introduced to:

CHAIN SHELL OS-9 COMMANDS

In Chapter Two we mentioned that an operating system is nothing more than a piece of software that lets you communicate — talk if you please — to many different types of hardware.

It is the operating system that lets your printer or disk file understand what you are saying on the keyboard. It is the operating system that gives you a way to hook your computer to another computer through a telephone line and modem.

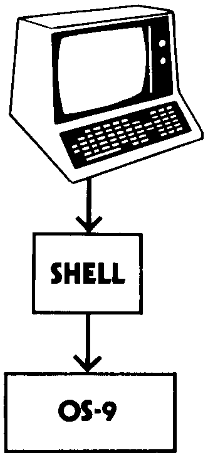
You'll find that with the OS-9 operating system, the possibilities are almost endless. You'll be limited only by the hardware you own and your imagination.

MEET THE OS-9 SHELL

When you talk to OS-9 by typing a command on your keyboard, you are communicating with the SHELL. The SHELL is a command interpreter that translates the words you type into an action by the computer. You'll know when you're talking to the SHELL because you'll see its prompt. It looks like this:

OS9:

When you see this prompt, you'll know that the SHELL is active and waiting for you to enter a command. To enter a command, you simply type a command line followed by a carriage return. You can use lower-case letters, upper-case letters or a combination — the SHELL doesn't care.



Now, let's take a closer look at an OS-9 command line. The first thing following the prompt should be the name of a program. This name can be the name of a program located in a module in your computer's memory or the name of a file that stores your program on a floppy disk.

The program itself can be 6809 machine code that executes directly, a module containing compiled intermediate code like that used by BASIC09, or a procedure file. Here's what happens when you type a program name in an OS-9 command line.

If the SHELL finds a module in memory with the name you have typed it will run the program. If it doesn't find the program in the module directory it looks for a disk file with that name in the current execution directory. If it finds the file, OS-9 loads it into memory and runs it.

If the name you typed is not the name of a module in memory or a file stored in the current execution directory, you still have another chance — it may be a procedure file. The SHELL knows this, and searches the current data directory for a file with the same name.

If the SHELL finds a file in the data directory, it assumes it is a procedure file and runs it. A procedure file is a special case. Instead of holding object code that runs on your computer, or I-code that is executed by BASIC09, it contains a text file that represents one or several command lines.

These command lines look just like the command lines you type on the keyboard. When the SHELL executes a procedure file, it reads the file one line at a time — as if it were reading data from the keyboard — and executes each command.

The program the SHELL reads from your keyboard or a procedure file is usually followed by one or more parameters. A parameter gives directions to the program that is being executed. It is separated from the program name by a space or spaces. For example, if you want to list a file called "secret" to your terminal you must type:

OS9:list secret (RETURN)

If you want it to be listed on your printer you type:

OS9:list secret >/p (RETURN)

In fact, you may even send the listing to another file:

OS9:list secret >copyofsecret (RETURN)

As we promised, OS-9 is a very versatile operating system.

Sometimes the parameters are options or modifiers. For example when you want to list the current data directory to your terminal, you type:

OS9:dir (RETURN)

This command line lists the names of all files in the current data directory to your terminal. But stand by, there's a way to get more information about the files. Try:

OS9:dir e (RETURN)

This command line lists all available statistics about each file in the current working directory. The "e" is an option that means list the "entire" directory record. Speaking of directories and options. If you want to see which files are stored in your current execution directory, try this:

OS9:dir x (RETURN)

Or, if you want to see all the information about the files you have stored in the current execution directory, use this command line:

OS9:dir x e (RETURN)

ALL ABOUT PATHLISTS AND DEVICE NAMES

When you used the **LIST** command above, you were using a filename as a parameter. In this case, the parameter was an abbreviated pathlist. Since you didn't pass any information about a device or directory, **LIST** assumed that the file was located in your current data directory. But, what happens when you want to access a file that is not located in your current data directory?

Don't give up. Entering complete pathlists is easy. A pathlist is a description of the complete route your data must take before it arrives at its destination. It may contain the name of a mass storage file, a directory file, or any Input/Output Device.

The authors of OS-9 chose the term "pathlist" instead of "filename" because in many cases you will be giving the SHELL a list that contains more than one name. For example, many "pathlists" contain a device name, and one or more directory names as well as the name of a data file. Each name in the pathlist is separated by a slash "/".

Here are the rules. Pathlists contain names that describe three things.

1. Names of Physical I/O devices
2. Names of Directories
3. Names of regular files

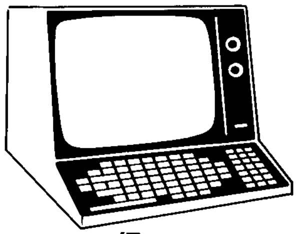
These names may contain as many as 29 characters or as few as one character. They must begin with either an upper-case or a lower-case letter. After that they may be made up of any combination of the following legal characters.

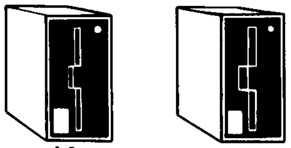
1. All uppercase letters: (A–Z)
2. All lowercase letters: (a–z)
3. The ten decimal digits: (0–9)
4. The underscore: (_)
5. The period: (.)

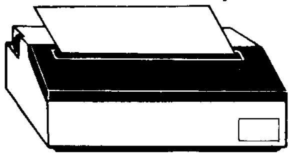
Here are some legal names:

the.BOOK
Chapter.one_and.two
XYZ123

Are you wondering how OS-9 can tell the difference between a filename and a device name? Here's the secret.


/Term




/p

A device name always starts with a slash, **/**. If the device holds multiple files — a disk drive for example — another slash followed by a directory or a filename usually follows the device name. If, however, the device cannot handle multiple files — as is the case with a terminal or printer — nothing follows the device name.

Here are the standard Microware device names:

NAME	DEVICE
TERM	Primary system terminal
T1,T2	Additional terminals
P	Parallel printer
P1	Serial printer
D0	Floppy Disk drive zero
D1	Floppy Disk drive one
H0	Hard Disk zero, etc.

Remember that if you want to name these devices in a pathlist, you must type a slash before their name. Here are some common pathlists.

Group 1: **/TERM** **/T1** **/p** **/p1**

Group 2: **/D0** **/d0/cmds** **/d1/worktext/the.book**

The pathlists in Group 1 refer to devices that cannot handle multiple files.

The pathlists in Group 2 are more complex. The pathlist **/d0** refers to disk drive number zero. If you needed to know the names of the files stored on this drive, you would use this command line:

```
OS9:dir /d0 (RETURN)
```

After you hit the [RETURN] key, the names of all files that you have previously saved on the disk installed in drive "d0" will be listed on your terminal. Let's try another command line:

```
OS9:dir /d0/cmds (RETURN)
```

This command lists the names of all files stored in a directory named **CMDS** located on the disk installed in drive **/d0**.

Let's take it one more step:

```
OS9:list /d1/worktext/the.book (RETURN)
```

This command prints a listing of a file named **the.book**. The file is located in a directory named "worktext" on the disk you have installed in device **/d1**.

To find out the details about installing and using directories please refer to your OS-9 Operating System Users Manual.

MORE ADVANCED FEATURES

OS-9 has many advanced features and we'll introduce you to a few in passing. For complete understanding, a thorough study of the OS-9 Users Manual is needed. Advanced features include:

1. I/O redirection
2. Memory Allocation
3. Multitasking

During normal operation all input for your program comes from OS-9's standard input path. Likewise, all output either goes to the standard output path or the standard error output path.

Reports, listings, and other data generated by your programs are usually sent to the standard output path. Error messages and various prompts are routinely sent to the standard error output path. All three paths normally lead to your terminal.

When you redirect the input you are telling OS-9 to get its input somewhere else. Likewise, when you redirect the output you are telling OS-9 to send its data somewhere other than your terminal. For example, when you sent the directory listing to the printer earlier in this chapter, you were redirecting the output to the printer.

There are three redirection operators that you will see in SHELL command lines:

<	means redirect the standard input path
>	means redirect the standard output path
>>	means redirect the standard error output path

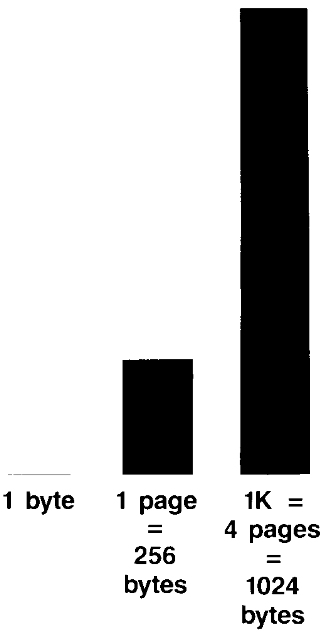
There are many ways to use these operators. You may redirect the input to your program from another terminal on the system or from a modem. Or, you may send output to a disk file for later printing. There is no end to the possibilities. You may even type on your line printer with this OS-9 command line.

```
OS9:echo /term >/p
```

Some OS-9 programs need very little memory to run. Others require thousands of bytes. This is not a problem, however, because the header of each program module tells OS-9 the minimum amount of memory needed to run a program. However, when you need more memory, it is an easy matter to request more with OS-9's memory size modifier. There are two ways to request more memory.

```
OS9:copy #8 myfile yourfile
OS9:copy #2K hisfile herfile
```

MEMORY UNITS



The first command line above instructs OS-9's copy utility to use eight 256-byte pages of memory — a total of 2048 bytes. And believe it or not, the second example also gives the copy command 2048 bytes to use. It is requesting two "K" or two thousand bytes of memory. For a detailed explanation of why 1K is actually 1024 bytes and 2K is actually 2048 bytes, see Chapter Seven.

There are also several ways to run a series of OS-9 programs. You can run them sequentially — one after the other; you can run them concurrently — all at the same time; or, you can synchronize them so that the output of one feeds the input of another using OS-9's pipes.

There are two ways to run programs sequentially. You may type one command line followed by a carriage return, wait for the program to finish and then type the next command line — or, you may type more than one command on a line. You must use a semi-colon to separate the commands if you chose the second method. Here's an example:

```
OS9: copy hisfile herfile ; dir >/p (RETURN)
```

This command line will copy the file named **hisfile** from the current data directory to a file named **herfile** in the same directory. It will then immediately print a listing of the current data directory on your printer.

If you want to run more than one program at the same time you must ask OS-9 to execute the programs concurrently by using an ampersand, **&**.

You may run any number of programs at the same time. The main restriction is usually the amount of memory in your system.

Pretend that you have just finished a book. You need to print a listing for your editor, but at the same time you need to be working on a term paper for a professor at the college course you're attending at night. To do both jobs at the same time, try this!

```
OS9:list mybook >/p&
&004

OS9:edit stinking_assignment
```

It's magic. Soon after you type the first line, your printer will start. Yet, the familiar OS-9 prompt will pop on the screen almost immediately. As soon as it appears you can type the command line that starts your editor. The printer will run as long as it needs to print the book without bothering you at all — if you can stand the noise.

LOGGING ON A TIMESHARING TERMINAL

With OS-9 you can do more than just print one file while you are editing another. In fact, one of the major uses for concurrent execution is terminal timesharing. For example, you could use your editor to write a news release about a new product on one terminal while your secretary is running the company payroll on another. Here's what it will look like.

ON YOUR TERMINAL:

```
OS9:tsmon /t1&
&005
OS9:
```

ON THE TIMESHARING TERMINAL

```
OS-9 Level 2 Version 1.0 Timesharing System 2/14/83 15:30:35
User name?: esther
Password:
Process #5 logged 2/14/83 15:31:47

Shell

OS9:
```

Isn't it amazing. Your terminal has prompted you to go back to work. While you're writing that news release, your secretary can probably finish the payroll and balance the books. Your computer will pay for itself in no time.

Usually, your timesharing terminal will be started automatically by a procedure file that runs when you turn on your computer. We showed you how to start it manually so you would know how it works.

When you first run the timesharing monitor program, **TSMON**, nothing happens. The terminal remains idle until someone hits its return key.

Also, when using a terminal on a timesharing system other than the master terminal, you must log on to the system. To log on you enter your name and the proper password. You will need to get your password from the system manager before you attempt to log on the first time. If you don't know this magic word, you won't be allowed on the system.

You have three chances to enter a valid user name and password. If you fail the test three times, the system will terminate the log-in sequence. If you are trying to use a system through a telephone and modem, you will most likely be disconnected. To log off an OS-9 system from a timesharing terminal, you need only hit the **ESCAPE** key representing an end-of-file signal on most systems and it returns your terminal to an idle state.

YOU CAN FEEL SECURE

If you work on a timesharing OS-9 system you needn't worry about someone else writing in your data files. The system protects you with its file security system.

Each OS-9 directory and file has several attributes that tell the system who owns the file and who may use it. They are:

1. Write permission for owner.
2. Read permission for owner.
3. Execute permission for owner.
4. Write permission for public.
5. Read permission for public.
6. Execute permission for public.
7. A "sharable" attribute.
8. A directory attribute.

Let's explain the special cases first. If the "sharable" attribute is turned on, OS-9 will not let two users use a file at the same time.

The directory attribute tells OS-9 that a file is a directory file. A directory file is special because it cannot be changed by the user. To change a directory or delete it during an operation would create total havoc with the file system. In fact, there would no longer be a system.

The other file security attributes almost explain themselves. They work because OS-9's file system automatically stores the user number associated with a process when it writes a file. If you are the owner of a process, you will own any files it creates.

If you **CREATE** a file with none of the public attributes set, you will be the only person that can **READ, WRITE,** or **EXECUTE** that file. You may even ask the system to protect a file from you. For example, after getting a mailing list in final form, you may clear both the public and owner **WRITE** permission attributes to prevent accidental deletion or modification.

CHAINING OS-9 COMMANDS TO BASIC09

There may be times when you want to exit BASIC09 and run an OS-9 utility command. The **CHAIN** statement lets you do it.

You'll most likely use **CHAIN** when you run a process that needs a large amount of memory. **CHAIN** frees a lot of memory because when it exits, it unlinks BASIC09 and returns all its memory to OS-9.

This means that if you want to return to BASIC09 after running an OS-9 utility you must use a sequential command line that executes BASIC09 as its last task. Here are a few examples:

```
CHAIN "ex BASIC09 menu"
CHAIN "DIR /D0/BASIC_Programs"
CHAIN "Dir; Echo*** Copying Directory***; ex BASIC09 copydir"
```

The first example exits BASIC09 and immediately executes it again. Then, BASIC09 immediately loads a procedure named **"menu"** from the current data directory and runs it.

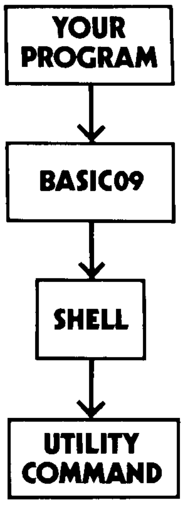
The second example exits BASIC09 and lists the directory **/D0/BASIC_Programs** to your terminal. After doing this, it returns control to the SHELL and you will see the familiar OS-9 prompt.

The final example, exits BASIC09, echos a message to your terminal, and then executes BASIC09 again. When it starts up the second time, BASIC09 automatically loads and runs a procedure named **copydir**.

Any files you have opened remain open when you run the **CHAIN** command. However, if you need to pass an open path to another program, you must use the **ex** option because of the way the SHELL handles data paths.

USING BASIC09'S SHELL STATEMENT

If you are running a small OS-9 program that doesn't need a lot of memory, you may use BASIC09's **SHELL** statement. It does not exit BASIC09, unlink and return memory. Rather, it puts BASIC09 to sleep temporarily.



The "Shell" Game

The **SHELL** statement lets you — or your BASIC09 procedures — access almost every OS-9 command. When you **RUN** it, the **SHELL** statement suspends BASIC09, and executes the OS-9 SHELL, passing the string expression you supply as a parameter.

If you need to exit BASIC09 temporarily to run a series of OS-9 commands, you may use the special "null string" case by passing an empty string in your command line.

When you pass this empty string, you will be greeted by OS-9's prompt immediately. You may then send as many commands as needed. When done, you need only hit the **ESCAPE** key to signal an end-of-file condition and terminate the **SHELL.** When the **SHELL** terminates, BASIC09 wakes up and you'll be back where you started. Let's look at a few examples.

```
SHELL "copy file1 file2"
SHELL "copy file1 file2&"
SHELL "edit Great_American_Novel"
```

The first statement calls the OS-9 **copy** utility and copies the contents of **file1** into **file2. File1** is assumed to be in the current data directory. **File2** will be created in the same directory.

The second example does the same job except it executes concurrently. This means that when the copy process is started, control returns to BASIC09. If the **SHELL** statement is in a procedure, the next line in that procedure will be executed immediately.

The last statement puts BASIC09 to sleep, calls the OS-9 SHELL and runs the system editor. It opens a file called **Great_American_Novel** in the current data directory.

OS-9 SYSTEM COMMANDS

You will find a complete description of each OS-9 utility command in Part Four of this book. After you start programming, you'll be able to use them directly from BASIC09 with the CHAIN and SHELL statements described above. We'll show you several sample command lines and where possible a sample run. You'll find the commands listed in alphabetical order for your convenience.

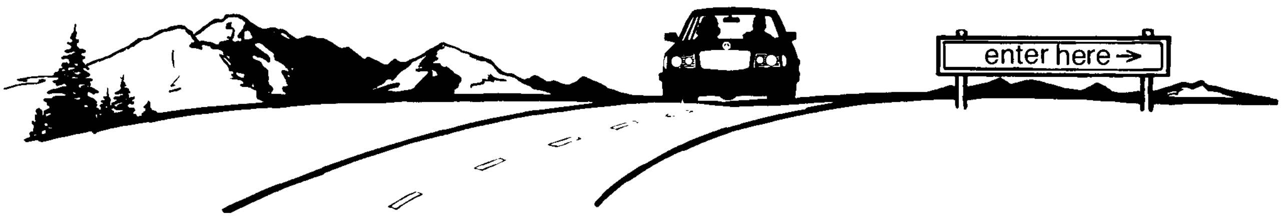
SUMMARY

In this chapter you have been introduced to one of the most powerful operating systems running on a microcomputer today, Microware's OS-9. You have also learned how to run operating system utility commands directly from BASIC09.

In the next Chapter we'll be getting down to business. You'll bring BASIC09 to life and learn how to enter and edit your own procedures. Make a quick pit stop now and we'll meet you in Chapter Four.

CHAPTER FOUR

BASIC09's editor



GETTING YOUR PROGRAM INTO THE COMPUTER

Did you remember to bring your learner's permit? I hope so, it sure would be a shame to get a traffic ticket while driving the Mercedes of computer languages.

You can't take a trip around the world without taking that first step out the door. Likewise with programming. You've arrived at a milestone — it's now or never. Be sure to keep your seatbelt fastened until you pass the bumps!

THE EDITOR

You could read about programming forever and enjoy it. But, it's more fun to program. Let's get started.

Before you can run even a simple program you must get it into your computer's memory. In this chapter we'll show you how to do that using BASIC09's editor. Then, as we move on to Chapter Five you'll have something to **LOAD, LIST, KILL,** and **RUN.** What was that saying your teacher drilled into your head back in the first grade? First things first?

Hold on tight! We should make 120 mph in this chapter as you learn about the following one and two keystroke commands.

(RETURN)	**+**	**+***	**–**	**–***
(LINE #)	**r**	**r***	**l**	**l***
(SPACE)	**s**	**s***	**d**	**d***
	c	**c***	**q**	

Don't panic. This business really does make sense. In fact, I'll bet you can see a pattern or two developing already.

Let's start at the beginning. First, bring BASIC09 alive by typing:

```
OS9: BASIC09 (RETURN)
```

In a few seconds you should see this message on your screen.

```
OS9: BASIC09

Copyright 1981 Microware and Motorola
Reproduced Under License
Basic09
B:
```

The **B**: means that BASIC09 is in its system mode and is waiting for you to tell it what to do. You'll be seeing this prompt a lot while you learn to program. Now, try typing:

```
B: edit (RETURN)
```

Remember, the **B**: is already on the screen. You only need to type **edit** and strike the key marked (RETURN) on your terminal's keyboard.

Shortly after you hit (RETURN), your screen should look like this:

```
B: edit
PROCEDURE Program
*
E:
```

After you typed **edit**, BASIC09 checked to see if you had typed a name for your procedure. Since you didn't, it named your procedure, **Program**.

Now type a **q** and you'll find yourself back in BASIC09's system mode. We're going to enter the editor again now. But this time we'll let you name your procedure. Don't let the word "procedure" scare you. On many BASIC systems a BASIC09 procedure would be called a "program." Here goes.

```
E:q
Ready
B:e screenfull (RETURN)
PROCEDURE screenfull
*
E:
```

Notice anything different? This time we didn't type **edit**. We were lazy and only typed the single letter, **e**. It worked though, didn't it? Stick with us. We'll show you a lot of short cuts.

We wanted to name our procedure screenfull, so we typed that name after the e. Again, the symbol (RETURN) means that you must hit the key marked, (RETURN) on your keyboard. When you strike the (RETURN) key you won't see anything appear on the screen, but the cursor will move to the left hand side of the screen and drop down to the next line.

Make sure you type the space between the **e** and your procedure name. If you don't, BASIC09 will get indignant and ask, **WHAT?**

On with the show. You're about to be introduced to the most important editor command.

IT'S ALL IN A (SPACE)

After our next example you'll understand what NASA has been selling all these years. As you enter your first program, hit the spacebar on your keyboard every time you see the symbol, [SPACE]. Go ahead, give it a try.

```
PROCEDURE screenfull
*
E:(SPACE) 100 print "BASIC09 is GREAT!!!"; (RETURN)
*
E:(SPACE) 110 goto 100 (RETURN)
*
E:q
Ready
B:
```

Congratulations! You've just typed your first program into a computer. You've also learned two new editing commands.

Here's what happened. Every time you typed a (SPACE) as the first character on a line, BASIC09 remembered what you typed. That's why we told you the (SPACE) was the most important editing command. Without it, you would wind up typing to yourself.

Did you notice that when you typed the letter **q** to quit and return to BASIC09's system mode, you typed it as the first character in the line. If you had typed a (SPACE) in front of **q**, the editor wouldn't have quit. It would have typed an error message. We'll show you one later.

There are two more details we can show you in this exercise. After you typed each program line you had to hit the (RETURN) key to tell the editor that you were finished with the line. We printed the (RETURN) symbol so that you would know exactly what to do. We'll drop it in later examples.

The Edit Pointer

```
FOR N = 1 TO 10
PRINT N
NEXT N
END
```

Did you notice that each time you hit the (RETURN) key, the editor printed a star, *, before it printed the prompt for the next line. That star shows you where the "edit pointer" is. What's an edit pointer? Let's explain.

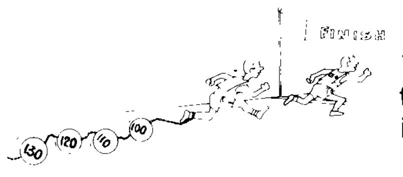
The edit pointer is how the editor keeps track of where the next command will work. It's like using your finger to keep your place when you read a book. Just as you might move your finger back and forth to re-read a paragraph, the edit pointer can also be moved back and forth.

After you hit the (RETURN) key, the editor entered your line into its memory and moved its edit pointer to the next available space in memory. It then printed a star and echoed the empty line. All you saw was the star. If the edit pointer had been moved to the first character of an existing line, BASIC09 would have printed the star, *, followed by the characters in that line.

Without this innocent looking star you wouldn't know your location in the program. Later on, when we show you how to insert a line in your programs, you'll realize the star's importance. For now, remember that a star in front of a line means that the editor is pointing to the first character in that line.

ABOUT THOSE LINE NUMBERS

We used line numbers in this first editing example to show you how they work and to give you a chance to get line numbers out of your system. Line numbers are a holdover from the days of dumb programming languages—modern languages such as Basic09, Pascal, C and others have little need for them, and for good reasons.



Here's some advice: if you don't have to use line numbers, don't. Your programs will be shorter. They will run faster. They will be easier to edit. And they'll be easier to read. Isn't that what programming is all about?

Here are a few facts you can use if you ever need to edit an existing program that uses line numbers. If you must enter a new line, type the line number followed by the program statement. The editor will automatically insert your line at the right place in the program because numbered lines are automatically stored in the computer's memory in ascending order.

When you need to move to a numbered line, you need only type the line number followed by a (RETURN). The editor will move to that line and print it. If the line number you requested does not exist, the editor will print the next higher numbered line. If you must delete a numbered line, simply move to it and then type the letter "d" in response to the prompt.

Here's another secret. You don't need to type the (SPACE) key before the line number when entering a program that uses line numbers.

We used the (SPACE) command in our example to show you how it works. You'll need it while entering most of our example programs, however. For the most part, the examples will not use line numbers.

RENUMBERING YOUR LINES

When you get stuck using an old program with line numbers, BASIC09 makes life easy. It even lets you renumber program lines. To renumber the lines in a program, you'll use a command statement like this.

```
E:r 1000,10
E:r* 1000,5
```

Notice that we had you type the **r** — an editor command — in the first position of the line. We did not tell you to use a (SPACE) command because commands must always be typed immediately after the prompt. Again, do not type a (SPACE) in front of a BASIC09 editor command. It will not work.

Did you remember to hit the (RETURN) key after you typed the line? No? How long did you wait before it dawned on you that something else was needed?

Let's explain the two commands on the previous page. The first starts at the current line and makes it line **1000.** Each line number after it is increased by **10**, i.e., your program will be numbered **1000, 1010, 1020, 1030,** etc.

When you run the **r** command you'll notice that all line numbers in front of the star (above the star on the screen) at the time you type the command, will not be renumbered. In fact, almost all of BASIC09's editing commands work from the star, *, toward the end of the program.

The second command, **r***, starts with the very first line number in your program and makes it line **1000.** The star you typed after the "r" makes this happen. Each additional line number is then increased by five. After you type this command line your program will be numbered **1000, 1005, 1010, 1015,** etc.

Here's another short cut. You may also renumber your program by simply typing one of these command lines:

```
E:r (RETURN)
E:r* (RETURN)
```

Notice that we typed the command lines without the beginning line number and increment. The editor will automatically use 100 as the first line number and increase each additional line number by 10. The 100 and the 10 are called "default" values. Essentially, typing the commands above will give you the same result as typing:

```
E:r 100,10
E:r* 100,10
```

There's one more thing you should know, and then we'll try to forget about line numbers forever. You don't need to worry about line numbers inside the program. For example, if you start with the following short program:

```
E:100 PRINT "Hello!"
E:110 GOTO 100
```

And renumber it with this command:

```
E:r* 1000,5
```

Your program will look like this:

```
1000 PRINT "Hello!"
1005 GOTO 1000
```

Line number **100** in the example above has been changed to **1000** by the renumber command. It was also changed inside line number **1005**. Remember, BASIC09's renumber command automatically changes all numbers inside a line that refer to a renumbered line.

MOVING THROUGH YOUR PROGRAM

There are five basic commands which you can use to move back and forth within your program. First, you'll need a program with a few more lines. Go ahead and type the program below.

Don't forget to type the (SPACE) as the first character following the prompt in each line. Also, do not type the numbers along the left edge of the listing. BASIC09 prints them automatically when it lists a procedure and we'll explain them in the next chapter.

Finally, don't worry about indenting while you type, because BASIC09 does indents for you automatically.

EXAMPLE NO. 2 KBAUD 7A

```
PROCEDURE kbaud7a
    0000  DIM reply,bell:STRING[1]
    0010  DIM answer,counter,loopcounter:INTEGER
    001F  DIM memory(5):INTEGER
    002B
    002C  PRINT CHR$($0C)
    0032  bell=CHR$(7)
    003A  INPUT "Ready?", reply
    0049  PRINT "Start!" +bell
    0058  counter=0
    005F  WHILE counter<1000 DO
    006C    counter=counter+1
    0077    answer=counter/2*3+4-5
    008B    GOSUB 280
    008F    FOR loopcounter=1 TO 5
    009F      memory(loopcounter)=answer
    00AB    NEXT loopcounter
    00B6  ENDWHILE
    00BA  PRINT bell+"stop!"
    00C7  END
    00C9  280 RETURN
```

Now that we have a program listing to use, let's go to work. First, answer the editor's prompt with:

E:L* (RETURN)

Congratulations! You've just been introduced to BASIC09's list command. We used an uppercase letter **L** in our example so you wouldn't confuse it with the number one. However, feel free to use a lowercase letter **l** when you program. BASIC09 recognizes it also.

See the indentation. It sure makes a listing look nice.

Notice the star that BASIC09 has been printed in front of the first line of the program. Good, let's move it.

Any of these commands will move the star from one line to another:

E :(RETURN)
E :+ (RETURN)
E :– (RETURN)

If you're lazy, you'll use the first example. One keystroke and you'll be on the next line. But, what do you do if you need to move more than one line at a time? You have several options. Try these:

E :+ (NUMBER) (RETURN)
E : – (NUMBER) (RETURN)

For example, if you need to move the edit pointer (star) down two lines you would:

a. hit the key marked, **+**
b. hit the **2** key, and
c. hit the key marked [RETURN]

After you hit (RETURN), BASIC09 will echo a star and the new line. To move backwards, you type the – key instead of the **+**.

Of course, you can move any number of lines at a time by changing the value of (NUMBER). For example, to move the star 10 lines you would type the number 10 in step "b." above. Go ahead and try it. It's a lot easier to do than it is to explain.

How would you like to be able to move the edit pointer all the way to the bottom of your program so that you can add a new line. Type:

E :+ * (RETURN)

What do you do when you need to move to the top of the program so that you may insert a comment? It's simple, just type:

E : – * (RETURN)

Take some time now to move the star around in the sample program. Remember, BASIC09's edit pointer is always located in front of the line printed behind the star. Once you master the star, you'll find the rest of this chapter a snap. Let's move on.

INSERTING A LINE

So you forgot to type a line. No problem, it's about time you joined the club. Review the section above. Got it? Ok, go ahead and move the star in front of the line that follows the line you forgot. Now use the (SPACE)command to insert the line you left out before. That was easy, wasn't it?

Let's look at an example. Suppose you want BASIC09 to print the message:

```
Hello
Space
Cadets
```

But you typed the program like this:

```
PRINT "Hello"
PRINT "Cadets"
```

Forgot something, didn't you? Never fret. Type:

```
E:-* (RETURN)
```

BASIC09 should move the star to the top of your program and your screen should look like this:

```
* print "Hello"
E:
```

This means that BASIC09's edit pointer is located in front of the first line of the program. You need to insert a line in front of the second line.

Looking at it from the editor's point of view, you want to insert a line in front of the line:

```
PRINT "Cadets"
```

Let's give it a try. Hit the (RETURN) key. BASIC09 should echo:

```
* PRINT "Cadets"
E:
```

Great! You have the star just where you want it. Do your thing. Type:

```
E:(SPACE) print "Space" (RETURN)
```

If you remember that BASIC09's editor always inserts lines in front of the line the star is pointing to, you should have it made.

You now have a program in memory which will print the message you wanted. But to prove it, you're going to have to learn another new editing command.

LISTING YOUR PROGRAM

It's time to get brave. Type:

```
E:L* (RETURN)
```

BASIC09 should list:

```
PROCEDURE Program
0000 PRINT "Hello"
0009 PRINT "Space"
0012 PRINT "Cadets"
*
E:
```

Congratulations. You now know how to list your procedures while editing.

Let's try something different. Move the star to the top of the procedure using the "-*" command. Then type:

```
E:L 2 (RETURN)
```

I'll bet BASIC09 echoed something like this:

```
PRINT "Hello"
PRINT "Space"
*
E:
```

Just as you can move any number of lines, you can list any number of lines. And don't forget, the **L*** lets you list an entire procedure. There's something magic about a star.

DELETING A LINE

Keep the faith. We're almost done. Besides, once you know how to edit programs you'll be able to play to your heart's content. Just for the fun of it, let's get rid of the word "Space" in the message above. Move the star in front of the line:

```
PRINT "Space"
```

Do you remember how to do it?

Now, type a **d** in the first position after the edit prompt. Don't be afraid. After you type the **d**, go ahead and list your program. Your screen should look something like this.

```
* print "Space"
E:d (RETURN)
* print "Cadets"
E:L* (RETURN)
print "Hello"
* print "Cadets"
E:
```

Just for practice, why don't you see if you can figure out how to put the line back in. Give it a try.

IN SEARCH OF A LOST STRING

When your procedures get longer, things will get a little tougher. It sure would be nice to be able to go directly to a BASIC09 statement or variable without looking through a long listing.

No problem! Once again BASIC09 comes to the rescue. It's time to learn how to use the Search command. The **PROCEDURE kbaud7a** which you typed in earlier will give you a few lines to work with.

Ready? Let's find the **WHILE** ... **DO** loop in the program. It's easy. Move the cursor to the top of the file. Then, type:

```
E:s .WHILE. (RETURN)
```

BASIC09 should respond by printing:

```
* WHILE k<1000 DO
E:
```

The periods, ., you typed on both sides of the word **WHILE** are delimiters. You may use any character as a delimiter — except a (SPACE) – if it is not present in the word you are looking for. The common punctuation marks work best. I use a period for a delimiter because it is easy for me to hit it on the keyboard. Many people use a slash, **/**. You should use the delimiter that works best for you.

If we had typed:

```
E:s aWHILEa (RETURN)
```

The search would have been successful. Personally, I think I'll stick to common punctuation marks.

When you use the **s** command without a star, the editor will start searching for the target string at the current position of the edit pointer. If it finds the string in a line, it will move the edit pointer (star) to the line and display it. If it can not find the string you are looking for, it will leave the edit pointer where it was and let you know. In the example above, it would print:

```
CAN'T FIND: "WHILE"
```

How would you like to find every occurence of a string in your procedure? It's a snap. Type:

```
s* .WHILE. (RETURN)
```

If you type the line right, BASIC09 should print:

```
WHILE k<1000 DO
ENDWHILE
E:
```

Notice that the star is positioned at the beginning of the last line in the procedure that contains the characters, **WHILE**.

How about that. Looks like it's time for another word about the star. Here's the secret. The star tells the search command to look for every occurence of the character string enclosed between the periods. It also tells the editor to print every line it finds that contains the target string. It leaves the edit pointer (star) in front of the last line containing the string.

And, aren't you lucky today? Here's a shortcut you can use with the search command. You don't need to type the second delimiter. Our search above would have worked if we had simply typed:

```
s .WHILE (RETURN)
```

CHANGING YOUR MIND — OR, FIXING AN ERROR

It's bound to happen. Sooner or later you are going to hit the wrong key while typing a procedure. It happens to me all the time.

When it happens to you, don't panic. We're going to show you BASIC09's "Change" command next.

Look at the listing of the procedure kbaud7a again. What do you think would happen if you typed **PRING** instead of **PRINT** in the fifth line? Try typing it that way and we'll take a look. Your screen should look something like this.

```
E: pring chr$($0C)
         ↑
pring chr$($0C)
      ↑
ERROR #027
— Missing Assignment Statement
* 002C ERR     pring chr$(0C)
E:
```

Whenever you make a mistake, BASIC09 will always report an ERROR. If you have the OS-9 utility PRINTERR activated, you will also receive an English language error report similar to the one above.

Notice that **pring** was not capitalized. If you had typed **print**, it would have been listed in all capital letters. The little arrow under the **c** in **chr$($0C)** was put there by the editor to help you find the error. Nice, isn't it?

Here's what happened in our example. The editor didn't recognize **pring** as one of BASIC09's reserved words, so it assumed that it was a variable that you wanted to use.

Still thinking that **pring** was a variable, the editor assumed that you wanted to assign the value of the expression, **chr$($0C)** to it. But alas, it couldn't find an equals sign, =.

At this point the only thing the editor could do was print the error message to let you know about your mistake. Of course, in all your infinite wisdom you immediately recognized that you meant to type, **print**. But how do you fix it?

The change command is easy to use and very similar to the search command above. To fix the mistake above you need only type:

```
E:c.pring.print. (RETURN)
```

The editor should immediately reply with:

```
* PRINT CHR$($0C)
E:
```

The **c** without the star looks for the first occurrence of your string — **pring** in this case. If it finds the string, it substitutes a replacement string. This time the replacement string was **print**.

Here's another secret. You really wouldn't have needed the period after the word **print**. The delimiter following the replacement string is optional. Deja Vu!

If you type the change command with a star, **c***, it really goes to work for you. In fact, every occurrence of your match string in the procedure is changed to your replacement string. This is extremely handy when you must change the name of a variable that you have used many times throughout a procedure.

It's time now for a word of caution. Be careful with the change command — especially if you are using the star. If your match string is short it could easily appear in a longer word.

Consider what would happen if you instinctively asked the editor to change **no** to **yes**. But, you forgot that one of your messages contained the word, **normal**.

That's right! After you ran the change command, **normal** became **yesrmal** — and that isn't.

You should have a pretty good handle on the editor by now. But, why stop. Go ahead and practice a few minutes. You'll probably amaze yourself. When you're finished, take a short break. You'll need the rest. We're predicting a fast track for our race through the BASIC09 system mode in the next chapter.

SUMMARY

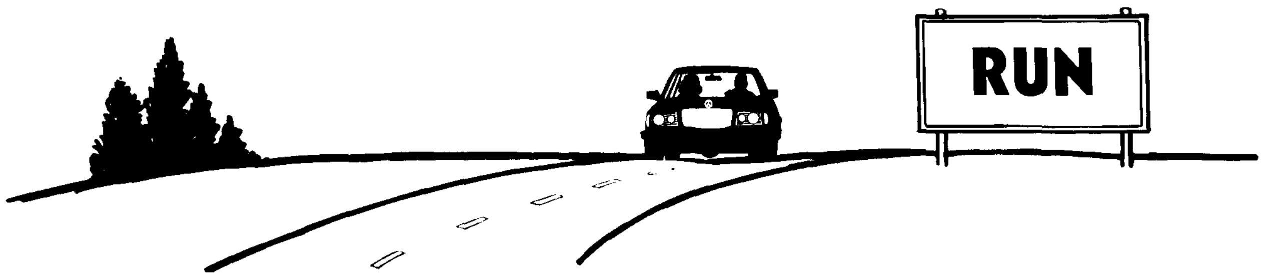
You should be in the drivers seat with BASIC09's editor. See you on the track.

In this Chapter, you have learned:

a. How to insert a line with the SPACE command.
b. How to enter a program with line numbers.
c. How to renumber a program.
d. How to move through your procedure.
e. How to list your pride and joy.
f. How to search for a string.
g. How to fix a mistake with the change command.

CHAPTER FIVE

how to run your programs



MANAGING YOUR PROGRAMS — THE SYSTEM EXECUTIVE

You say you bought your computer to solve problems. But, when you brought it home you had a bigger one. No one told you that you had to write programs and run them to solve problems. Is that what's getting you down? Hang in there.

In Chapter Four, we showed you how to enter a program into your computer's memory. Now, we're going to show you how to run it. You'll also learn how to **LOAD, LIST, SAVE, PACK,** and **KILL** it.

We'll show you how to request more memory, change your execution and data directories, rename your procedures, and find out what's in your workspace. As an extra added attraction, we'll run an OS-9 system utility.

Here's a list of the BASIC09 commands you'll learn in this chapter:

$	**BYE**	**CHD**	**CHX**
DIR	**EDIT**	**KILL**	**LIST**
LOAD	**MEM**	**PACK**	**RENAME**
RUN	**SAVE**		

THE BASIC09 WORKSPACE

Remember the phrase, "You can't see the forest for the trees." Let's see if we can put things in perspective.

OS-9 is an operating system that gives your computer the ability to talk to the real world. It lives in your computer's memory. You learned a few of its tricks in Chapter Two and were introduced to its power in Chapter Three.

BASIC09 is an interactive programming language. OS-9 loads it into memory when you ask for it from your terminal. When OS-9 brings BASIC09 to life it gives it a memory allowance, called a workspace.

I'll bet teenagers can identify with this concept; always wanting more "space" and more allowance? Do they always want them in that order?

GETTING MORE MEMORY

Let me show you how to quit when you get tired. Then, we'll park our Mercedes in the middle of the workspace and see what we can learn.

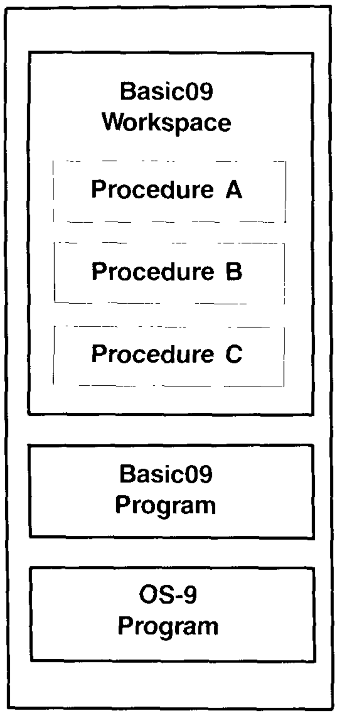
No, you can't just throw up your arms and quit. If you are going to be a successful programmer, you must learn to exit gracefully.

Besides, quitting is as simple as a three-letter word. Just answer BASIC09's system prompt with the three letters, **bye,** and control will return to the program that called BASIC09. Most of the time you'll see OS-9's prompt. On your terminal, it should look like this:

```
B:bye (RETURN)
OS9:
```

If you would rather only hit one key to exit BASIC09, just type the (ESCAPE) key—sometimes it's marked (ESC)— immediately after the system mode prompt, **B**:. You should see the familiar OS-9 prompt, **OS9**:.

THE COMPUTERS MEMORY



You didn't really want to quit already, did you? OK then, let's continue.

When you load BASIC09 from OS-9, you automatically receive a 4K workspace. The "4K" is computer talk which means about 4000 bytes of memory.

'What's a byte," you say? Try to picture a tiny mail box inside your computer. It's small and will only hold one character. Got the picture? You've just visualized a byte.

If you want to get really technical, the paragraph above isn't quite true. You actually receive 4096 bytes of memory in your workspace when you load BASIC09. It just happens that 4000 is the closest decimal number, rounded to the nearest 1000, of two raised to the 12th power.

Don't get illusions of grandeur too soon. BASIC09 uses 1280 bytes out of the 4096 to store its own data. This leaves you with 2816 bytes for your procedures and the variable storage you need to run them.

If you don't have enough memory in the workspace to hold both your program and its variables, BASIC09 will refuse to run your program. Talk about independent computers! Don't worry. There's almost always an easy answer in BASIC09.

There are two ways to get more memory. If you think ahead, you can ask for it when you call BASIC09. To get more memory you use an OS-9 command line like this.

```
OS9:BASIC09 #20K (RETURN)
```

This command gives you a workspace approximately 20,000 bytes long. The actual count is 19,200 (20,480 less BASIC09's reserved 1,280) if you're a stickler for details. And if you want to be a programmer, you had better be.

If you're like most of us, you'll forget to ask for enough memory. What then? Again, no problem. Just use BASIC09's **MEM** command.

There are two ways to use **MEM**:

```
B:MEM (RETURN)
B:MEM 8000 (RETURN)
```

When you use the first example, you are asking BASIC09 a question. 'How much memory do I have in my workspace?'' Your conversation should look like this:

```
B:mem
4096
Ready
B:
```

If you type a decimal number after **MEM** like we did in the second example, you are sending **BASIC09** a message. You are saying, ''Hey BASIC09! I need more space! Give me 8000 bytes.''

CHECKING AND CHANGING DIRECTORIES

Let your fingers do the walking. Sound familiar? I haven't seen a yellow keyboard yet but I'd be hard pressed to find a better analogy. When you need to know someone's phone number, you look it up in the telephone directory — unless you're lazy and call information. Asking BASIC09 to see what is in the workspace is easier than calling information.

The **DIR** system command will show you the name of all procedures in your workspace. It reports their size and tells you how much memory they need for data storage. **DIR** also tells you how much memory remains available in your workspace and marks all PACKed procedures with a dash, **-**.

DIR may be called to action in many ways. Here are a few:

```
B: (RETURN)
B:dir (RETURN)
B:dir >/p (RETURN)
B:dir >mydirlist (RETURN)
```

The first and second examples print a directory listing on your terminal. If your workspace is empty and you haven't asked for extra memory, your exchange should look something like this.

```
B:dir
Name          Proc-Size          Data Size
2816 free
Ready
B:
```

Now, let's look at the third command line. The greater than sign, >, is being used to redirect the output of the **DIR** command from the standard output to the printer.

DIR's output can be redirected to any valid OS-9 output device. For example, it could be sent to a disk file just as easily. In fact, that's what happens when you type the fourth command line. In this last example, the list of procedures in your workspace is written to a file called ''mydirlist'.

If you load the **PROCEDURE kbaud7a**, which you used to practice editing, and then run **DIR** you should see:

```
B:dir
Name              Proc-Size       Data-Size
* kbaud7a            281              42
2535 free
Ready
B:
```

You get quite a bit of information from **DIR**. Do you understand what it all means? Let's take a closer look.

The report above shows that your procedure is named **kbaud7a**. The **281** means that it is 281 bytes long. The **42** tells you that it needs 42 bytes of memory to store its data. The report also lets you know that you have 2535 bytes of memory left in the workspace. This memory may be used to store additional procedures or data. We'll talk about that sneaky star again soon.

THE ALL MIGHTY $

If **DIR** gives you the report above, what do you think would happen if you typed, **$dir**. Go ahead and try it. Your disk drives came to life, didn't they?

Here's what happened. You temporarily left BASIC09 and called the OS-9 SHELL because you typed the dollar sign, $.

Since you followed the dollar sign with the name of one of OS-9's system utility commands, **DIR**, the SHELL read the name and executed the command for you. When it finished, it returned you to BASIC09's system mode. Confused about **DIR**s? Don't worry, it can be confusing. The Basic09 **DIR** is completely different than the OS-9 **DIR** command. There are some other cases where command names in Basic09 are the same as OS-9 command names, so be careful because they do entirely different things!

The dollar sign command we're using here is very important. It is your door to OS-9 from inside BASIC09. It lets you run any OS-9 utility. Or, any machine language program that runs under OS-9 for that matter.

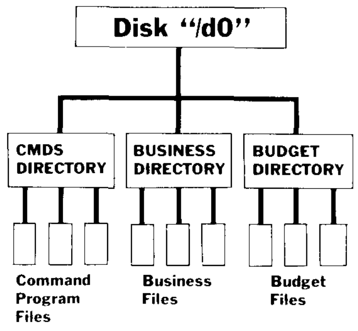
You can even create concurrent processes with the dollar sign command. This is a fancy way of saying that you can make your computer do more than one thing at a time.

For example, you can use the dollar sign command to tell OS-9 to print your annual report while you are making some last minute changes in a BASIC09 program. Of course, you must have enough memory in your computer to hold both programs. Doing two or more things at once is called multitasking. Remember?

CHANGING YOUR DIRECTORIES

What's that old saying? 'The grass is always greener on the other side of the fence.'' It's sometime true with directories too. Let's explain.

First, you must understand that OS-9 uses a tree-structured, or "hierarchical" directory system. Each disk can have more than one directory, and a directory can include the names of other directories. It lets you put related files together.



You can store your business records in files in a directory called **BUSINESS**. You can put your household records in a directory called **BUDGET**. You won't need to look through a long directory listing of household budget programs while you're trying to find a file containing information about last year's sales.

OS-9 always uses at least two directories. The first is called the "execution directory" which has the name **CMDS** for "commands". It holds command files that can be run by your computer. You will find OS-9 utility commands in the execution directory. This directory also holds **PACK**ed BASIC09 modules. We'll tell you more about these later in this chapter.

The data directory is used to hold all kinds of data. BASIC09 procedures are a good example. When you **SAVE** a procedure from BASIC09, it is automatically filed in the current Data directory. If you are working on a timesharing system, your data directory usually has the same name as your user name. On single-user systems, it is usually the main directory of an entire disk drive such as **/D1**.

There are two directory change commands — **CHD** and **CHX. CHD** is used to change the current Data directory. **CHX** is used to change execution directories. Here are two examples.

```
B:chd /d1/budget (RETURN)
B:chx /d0/basicx (RETURN)
```

The first command changes the current data directory to a directory file named **BUDGET** located on device **/d1**. The second changes the current execution directory to a directory on device **/d0** named **BASICX**.

TO "E" OR TO "EDIT"

We are not being redundant lightly. We wanted this chapter to be complete and we didn't want you to forget that any time you are operating in BASIC09's system mode you may create or edit a procedure by typing either a single **E** or the word **EDIT** followed by the name of the procedure. Let's try it again.

```
B:edit futures
```

When you type the command above, BASIC09 will go into its edit mode and create a procedure named **FUTURES**. When you are finished with the editing session, you may type **SAVE** and your procedure will be stored automatically in a file named **FUTURES** in your current data directory.

SAVING YOUR PRIDE AND JOY

If you forget everything else in this chapter, remember the **SAVE** command. It will save a lot of wear and tear on your typing fingers. As you must have guessed by now, **SAVE** lets you write a copy of procedures to a file or device. Let's look at a few command lines.

```
B:save
B:save myprogram
B:save aprog bprog cprog
B:save aprog bprog cprog >programs
B:save*
B:save* bunchafiles
```

Stand by to see more about stars. That infamous asterisk is back again.

If you type the first command line above, BASIC09 saves the current working procedure. This is the last procedure named by a previous command. For example, if you have just finished editing a procedure named, **MYPROGRAM**, then that procedure is the current working procedure. The same procedure would be saved if you had just finished **LIST**ing **MYPROGRAM** or **LOAD**ing **MYPROGRAM**.

Don't forget, if you type in a procedure and fail to give it a name when you call the editor, BASIC09 will give it the name, **PROGRAM**.

What happens if you can't remember which procedure you worked on last? No problem. Type **DIR** (RETURN) — or just (RETURN) and look for the procedure marked with a star. Try it.

If you type the second command line above, BASIC09 will look for a procedure named **MYPROGRAM** in your workspace and **SAVE** it in the current data directory. It will be filed with the same name, **MYPROGRAM**. If BASIC09 is unable to find the procedure you will soon see an error message on your screen.

The third example **SAVE**s the procedures, **APROG, BPROG,** and **CPROG** in a file named **APROG** in the current data directory. The fourth example **SAVE**s the same procedures as the third. However, it **SAVE**s them in a file named, **PROGRAMS** in the current data directory.

The next to last command line **SAVE**s every procedure in your workspace. The file is stored in the current data directory and will have the same name as the first procedure in the workspace. The last example **SAVE**s every procedure in the workspace also, but it stores it in a file named **BUNCHAFILES**.

Here's a short procedure that you can type in and **SAVE** for practice. It's also a useful program that you can use to figure out how to invest the money you make writing powerful applications programs in BASIC09.

EXAMPLE NO. 3: FUTURVAL

```
PROCEDURE futurval
 0000   DIM value,principal,interest:REAL
 000F   DIM timescompounded,years:INTEGER
 001A
 001B   PRINT "Let's figure the future value of an investment!"
 004F
 0050   REM First we need to ask a few questions.
 0078
 0079   INPUT "What is your initial investment?",principal
 00A2   INPUT "OK, What is the nominal interest rate?", interest
 00D1   INPUT "How many times will interest be compounded?",timescompounded
 0105   INPUT "How many years?",years
 011D
 011E   interest=interest/timescompounded/100
 012F   value=principal*(1+interest)↑(timescompounded*years)
 0148
 0149   PRINT "Your future value is"; INT(value*100+.5)/100
 0177   PRINT
 0179
 017A   END
```

It's interesting to note that when you **SAVE** a program you are merely **LIST**ing it to a disk file. Yet, there is a difference in the files produced by the **SAVE** and **LIST** commands. **SAVE** writes a straight file to your disk. There is no indentation or other pretty printing.

Remember, BASIC09 formats your **LIST**ings so that loops and other control structures are indented to enhance readability. You should also be aware that if you tell BASIC09 to **LOAD** a file of procedures that have been **LIST**ed to a disk file, you will get nothing but errors. Go ahead and try it if you feel brave.

The bottom line! If you plan on **LOAD**ing a file later. You must **SAVE** it now.

LOADING YOUR FILES

Here's an easy command for you. **LOAD** does just what its name implies. For example:

```
B:load myprogram
B:load /d1/programs/myprogram
```

The first command **LOAD**s the file **MYPROGRAM** from the current data directory into your workspace. The second **LOAD**s the file **MYPROGRAM** from a directory named **PROGRAMS** on device **/d1**.

Files may include one procedure. Or, they may include many procedures. What you **SAVE** is what you **LOAD**.

RENAME YOUR PROCEDURES

Here's another easy one. Tired of the name you gave a procedure when you first edited it. Don't fret. **RENAME** it. Here's the format:

B:rename xyz abc [RETURN]

If you run the command above, BASIC09 will look in your workspace for a procedure named **XYZ**. If it finds it, it will name it, **ABC**. If the procedure is not in the workspace, you'll find yourself reading an error message on your terminal.

Make sure you know the difference between **RENAME** and **$RENAME.** **RENAME** is a BASIC09 system command that will rename a procedure in your BASIC09 workspace. **$RENAME**, on the other hand, will cause your computer to execute the **OS-9 RENAME** utility. It is used to rename disk files.

Remember the dollar sign. You may never get rich, but you'll know how to use all the powerful OS-9 utility commands.

RUNNING YOUR PROGRAM

Paydirt! You have arrived. This is why you bought your computer. And, it's almost as simple as a three-letter word.

Before you try the **RUN** command, type in the following procedure using BASIC09's **EDIT**or. Leave the procedure **FUTURVAL** in your workspace.

EXAMPLE NO. 4: ROMAN

```
PROCEDURE roman
 0000   (* prints integer parameter as roman numeral
 002D   (* data must be >0
 003F
 0040   PARAM number:INTEGER
 0047   DIM value,svalu,position:INTEGER
 0056   DIM char,subs:STRING
 0061
 0062   char:= "MDCLXVI"
 0070   subs:= "CCXXII"
 007E
 007F   PRINT \ PRINT
 0083   DATA 1000,100,500,100,100,10,50,10,10,1,5,1,1,0
 00B3
 00B4   FOR position=1 TO 7
 00C4       READ value
 00C9       READ svalu
 00CE
 00CF     WHILE number>=value DO
 00DC       PRINT MID$(char,position,1);
 00E8       number:=number-value
 00F4     ENDWHILE
 00F8
 00F9     IF number>=value-svalu THEN
 010A       PRINT MID$(subs,position,1); MID$(char,position,1);
 0120       number:=number-value+svalu
 0130     ENDIF
 0132
 0133   NEXT position
 013E   PRINT \ PRINT
 0142   END
 0144
```

HERE'S A QUIZ

Decimal	Roman?
6809	
68	
10	
25	
199	

Now's the time to get into a good habit. As soon as you have finished typing your new procedure into the workspace, use the **SAVE** command to write it to a disk file. Typing it over again wouldn't be that much fun!

You should now have two procedures in your workspace. One should be named **FUTURVAL**, the other **ROMAN**. Let's run them.

```
B:run (RETURN)
B:run futurval (RETURN)
B:run roman(1983) (RETURN)
```

Go ahead and try the first command line. What happened? I'll bet it reported an error. Can you guess why?

When you typed **run**, BASIC09 tried to **RUN** the procedure **ROMAN** because when you do not specify a procedure name, BASIC09 automatically runs the procedure with the star. In this case it was **ROMAN** since you just finished editing and saving that procedure.

Despite all of your good intentions, you didn't give BASIC09 all the information it needed to run the procedure. You left out a parameter. Don't worry about the big word. We'll cover it thoroughly in Chapter 12 when we show you how to let your procedures run other procedures.

Now try the second command line. BASIC09 should run the procedure **FUTURVAL** successfully. Did you have fun dreaming about the money you're going to make? How about a loan?

Now type the first command line again. BASIC09 should have run **FUTURVAL** again. It did because **FUTURVAL** is now the procedure with the star. Got the idea? Good! Let's move on.

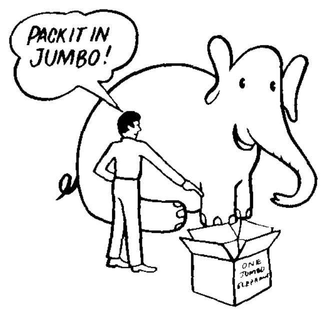
Type the third command line. I'll bet you were shocked when BASIC09 printed 1983 in Roman numerals that fast. You haven't seen anything yet. Wait till you see it run your own program.

Time out for a technical note. If BASIC09 can't find a procedure you request in your workspace, it won't quit. It will automatically look outside the workspace for a module of the same name. If it finds one, and it is a packed procedure, it will run it.

That's not all. If BASIC09 doesn't find a module, it still won't quit. It will look in the current execution directory and try to find a file with the same name. If it finds the file, it will automatically load it and attempt to execute it.

Go ahead and practice with the three examples of the run command above until you are sick of Roman numerals and tired of investing. Then, we'll continue our tour. Leave the two procedures in your workspace when you are through. We'll need them in the next section.

TAKE THIS PROCEDURE AND PACK IT!



Suppose you have just written the hottest computer program going. You want to sell it, but you don't want to give away your secrets. What do you do? With BASIC09 you can **PACK** it.

The **PACK** command tells BASIC09's compiler to make an extra pass on your procedure. When it's finished, there will be a whole new ball game. Variable names will disappear. Line numbers will be gone. **REM**arks will be removed. Your program will be smaller, faster and won't take up any room in your workspace.

After you **PACK** a procedure, no one will be able to list your program. That's just what you wanted, isn't it?

But there's a catch. After you **PACK** a procedure, you won't be able to **LIST** it either. No problem. Just make sure you **SAVE** it with the **SAVE** command before you **PACK** it.

When you forget, you will learn an important lesson the hard way — you must always **SAVE** a procedure before you **PACK** it. **PACKed** procedures can not be listed.

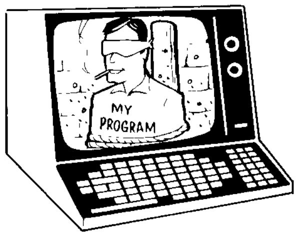
Here's a sequence of commands that will allow you to **SAVE, PACK,** and **RUN** the procedure **FUTURVAL.**

```
B:save futurval (RETURN)
B:pack futurval (RETURN)
B:kill futurval (RETURN)
B:run futurval (RETURN)
B:kill futurval (RETURN)
```

The first command will **SAVE** the procedure. The second will **PACK** it and save it in a file in the current execution directory. The third will remove it from the workspace. The fourth will load the file from the current execution directory outside the BASIC09 workspace and **RUN** it. The last command line will remove the procedure from memory outside your workspace.

You do not need to **KILL** a procedure outside the workspace immediately if you plan to use it again. However, if you are not going to use a procedure again, you should **KILL** it, because it takes up precious memory in the system.

We should note here that the format or syntax of the **PACK** command is just like that of the **LIST** and **SAVE** commands. In fact, it would be a good idea to review the paragraphs which cover the SAVE command now. You'll soon remember how to apply the "star" and redirect your files.



KILLING A PROCEDURE

The next command is deadly. It destroys procedures and removes them from your workspace. Again, make sure you **SAVE** any procedures you think you'll need later — before you **KILL** them. Here's the format.

```
B:kill myprogram
B:kill*
```

The first example **KILLs** only the procedure **MYPROGRAM**. The second **KILLs** every procedure in your workspace. Use it with caution.

SUMMARY

You've been busy and deserve a rest. I'll bet you can even speed shift this Mercedes by now. Why don't you reward yourself with a quick break. Then, join us in Chapter Six where we'll show you how to find the pesty parasites that keep trying to sneak into your programs.

In this chapter you have learned about:

a. Exiting BASIC09
b. Asking for more memory
c. Looking at directories
d. Changing directories
e. Saving procedures
f. Loading procedures
g. Listing procedures
h. Renaming procedures
i. Running procedures
j. Packing procedures
k. Killing procedures

CHAPTER SIX

debugging your programs

WHAT IF I MAKE A MISTAKE: AN ODE TO DEBUG

There comes a time in life when you realize that you aren't perfect.

To some, this realization comes early. To others, it comes late and is a shock. But if you're learning programming, it will most likely come as soon as you type, **RUN**.

In Chapter Five we showed you how to **LOAD, RUN, LIST, SAVE, PACK,** and **KILL** your programs. Since we didn't want to discourage you, we didn't mention debugging. May those carefree days rest in peace. Let's move on.

In this chapter we'll show you how to set **BREAK**points and change from **DEG**rees to **RAD**ians. We'll also **LET** you set the value of a variable, check the **STATE** of your program, **STEP** through a procedure one line at a time, and turn the **TRACE** mode on or off.

Here's a list of BASIC09's Debug Commands:

$	**DEBUG**	**LET**	**Q**	**STEP**
BREAK	**DIR**	**LIST**	**RAD**	**TROFF**
CONT	**END**	**PRINT**	**STATE**	**TRON**

DEBUGGING STARTS AUTOMATICALLY

I'll bet you've been worrying since Chapter Five when we didn't list **DEBUG** as an available command in the system mode. Now you can relax — BASIC09 was designed to make your life easy. After all, there's no need to debug a program that runs. Remember dad's advice — "If it works don't fix it."

So what happens when you do make a mistake — despite all your tender loving care — while coding? Rest easy, BASIC09 won't even say, "I told you so." Rather, it will enter the Debug mode automatically. That's why we didn't tell you about it in Chapter Five.

Once BASIC09 has activated its debugger, you'll have a number of powerful commands at your fingertips to help you trap the pesty parasites that try to hide in your code.

FORCING BASIC09 TO ENTER DEBUG

There will be times when your program refuses to cooperate. It will **RUN** fine — going through all the motions. It won't even crash and enter the Debug mode. Yet, it will refuse to compute the correct answer. What then?

No problem, that's when you force it to enter the Debug mode so you can look for the problem. You can do this in two ways.

The quick and dirty way is to type a **CONTROL C** on the keyboard. Remember, to do this you must hold down the key marked **CTRL** while you strike the **C**. When you hit this key, you generate a keyboard interrupt that causes BASIC09 to stop execution and enter the Debug mode. You'll know you've entered the debugger because you'll see a new prompt, **D**:.

When you stop a program by typing **Control C** on your keyboard, you should see a message like this:

BREAK PROCEDURE Program
D:

The other way to enter the Debug mode is to insert a **PAUSE** statement in your procedure using BASIC09's editor. Your procedure will run normally until the **PAUSE** statement is reached. Then, when it arrives at the line containing the **PAUSE,** it will halt and automatically enter the debugger to let you stop at a known line and check the value of suspected variables, etc.

Oh, I almost forgot. If you're afraid that you may not be able to remember why your program entered the debugger, you can have the **PAUSE** statement print a reminder for you. For example:

PAUSE "STOPPING SO YOU CAN CHECK THIS LOOP"

You can keep BASIC09 from entering the Debugger automatically when you know there is going to be an error. A good example is when you reach the end of a file while reading data from a disk.

Use an **ON ERROR GOTO** statement in your procedure when you know there will be an error generated to let BASIC09 handle the error without exiting your procedure. This practice is called "error trapping" and will be covered in greater detail in Chapter Nine.

Remember the trick you played while you were learning about system mode — the time you escaped OS-9 and had the system print a report while you were putting the finishing touches on a BASIC09 program? You typed a dollar sign followed by a valid OS-9 command line.

```
B:$List secret
```

This line loaded OS-9's **LIST** utility command into memory and then blatantly printed the contents of your **secret** file on the screen while the whole office was watching. That's the bad news.

The good news is that you can do the same thing from the Debug mode. You can even escape from Debug to play a quick game of Trek when you get tired of chasing bugs. Only the prompt line will be different. It should look like this.

```
D:$trek
```

Here's some more good news. Two commands, **DIR** and **LIST**, work like they did when you learned them in Chapter Five. You can only **LIST** the current procedure, however. If you need a refresher you can read about these commands in Chapter Five.

Back to work! Load a procedure into your workspace and we'll let you experiment. Try this one.

EXAMPLE NO. 5: INTEGERSUM

```
PROCEDURE integersum
  0000  total=0
  0008  FOR counter=1 TO 1000
  001B    total=total+counter
  0027  NEXT counter
  0032  PRINT "The sum of the first 1000 integers is"; total
  0060  END
```

Have you ever wondered about the sum of the first 1000 integers? Wouldn't you like to know that total so you can spit it out like an IBM-370 at the next office party? Hold on to your terminal, type **RUN** and we'll find out together.

That's a big number, isn't it. Let's have some more fun. Using Debug, we'll watch BASIC09 compute the answer.

To get started, enter the Edit mode and insert the word **PAUSE** as the first line of the program. Then, exit to the system mode and type **RUN**. What do you see? You should see a message like this:

```
BREAK PROCEDURE integersum
D:
```

If you want to see that number again, type:

```
D:CONT (RETURN)
```

As soon as you hit the return key the procedure will **CONT**inue like nothing ever happened. What would you do with 500500 widgets?

I QUIT: OR, HOW TO END A DEBUGGING SESSION GRACEFULLY

Before we get in too deep, we better show you how to exit a Debugging session gracefully. Besides, its as simple as a three letter word. Or, a one letter word if you're really exhausted.

When you are ready to give up or have solved the problem, answer the **D**: prompt with one of these command lines:

```
D:END (RETURN)
D:Q (RETURN)
```

These commands will stop the execution of all procedures in your workspace. They will close any input/output paths that are open and return you to the system mode. You'll know you're home free when you see the familiar **B**: prompt.

TRACE ON AND TRACE OFF

Moving right along, we'll turn BASIC09's trace mode on and let you watch the program execute step by step. Your conversation with the debugger should look like this:

```
B:RUN (RETURN)
BREAK: PROCEDURE integersum
D:TRON (RETURN)
D:STEP (RETURN)
BREAK: PROCEDURE integersum
D:(RETURN)
*0002    k=0
 =.0
D:(RETURN)
*000A    FOR n=1 TO 1000
 =1.
 =1000.
D:(RETURN)
*001D    k=k+n
 =1.
D:(RETURN)
*0029    NEXT N
 =2.
D:(RETURN)
*001D    k=k+n
 =3.
D:
```

When you typed **TRON**, BASIC09 turned the trace mode on within the **PROCEDURE integersum**. You're telling the system, "Hey! I want you to tell me what's going on."

TRON then goes to work. First, it decompiles a line of your code so that it can display the source statement you used when you wrote the procedure. After it prints this line of source code on your terminal, it executes the statement. If an expression is evaluated in the statement, the result of that evaluation is printed on the next line. You will see an equal sign followed by the result.

TRON prints one line for each expression that is evaluated. If you are tracing your program in the **STEP** mode, as we are above, the debugger will print the **D**: prompt after evaluating and executing each line.

Study the sequence above, or better yet, **RUN** it yourself. You'll soon understand what makes the program tick. You'll also soon grow tired of following a loop like the one in the **PROCEDURE integer-sum.** After all, that **FOR/NEXT** loop executes 1000 times every time you run the procedure.

TAKING BIGGER STEPS

That's right. We have an answer to your boredom. Take bigger **STEP**s. The secret lies in the syntax of the **STEP** statement.

When you simply type, **STEP** (RETURN) — like you did in the example above — you are asking the debugger to single-step through your code. You could have gotten the same result by typing, **STEP 1** (RETURN) .

"But if I can type **STEP 1**," you ask, "what's stopping me from typing **STEP 100**?"

Absolutely nothing. In fact if you type **STEP 100**, the debugger will execute 100 source statements before stopping. Of course if you have turned the trace mode on with a **TRON** statement, it will also print the results of any evaluation in each line it executes. If there are enough trees in the forest to make the paper, you can print a trace of every instruction in your program. If you have enough time to look through the listing you can find out where your code went astray.

There's a better way. Use the editor to enter **TRON** and **TROFF** statements at different places in your procedure. Then you can print an evaluation, enter a loop and run around in circles a few hundred times and exit to print another evaluation. If your program never exits from the loop, or it comes out displaying the wrong answers, you've probably found your problem.

In case you hadn't guessed, **TROFF** is the opposite of **TRON**. It turns the debugger's trace mode off.

Here's an interesting sidelight you should know about. If you turn the trace mode on in one procedure and that procedure calls another, the trace will be off while BASIC09 is executing the called procedure. In other words, **TRON** is local to a procedure. If you need to know what is happening in the called procedure you can easily add a **TRON** statement there.

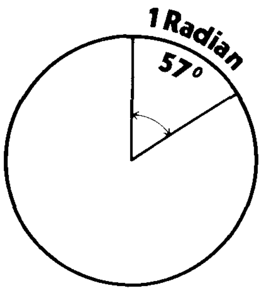
SETTING BREAKPOINTS IN YOUR PROGRAM

Quite often you'll want to set breakpoints in your program so that you can examine variables. You can do this while in the Debug mode by typing a simple command. For example:

```
D:BREAK integersum (RETURN)
```

This command will cause execution of your program to stop when the **PROCEDURE integersum** is entered. When it stops you will see the "D:" prompt and will be free to use all the BASIC09 debugging commands.

DEGREES AND RADIANS



For example, if you are working with a mathematical procedure that is manipulating angles, you may want to change the way BASIC09's math routines view them. The **SIN, COS, TAN, ACS, ASN,** and **ATN** functions can deal with both degrees and radians. But, they must know which game they are in before they start.

If the computer assumes you are giving it degrees and you are actually sending radians, you are in real trouble. While there are 360 degrees in a circle, there are only 6.28 radians. One radian is equal to 57.295 degrees.

BASIC09's authors didn't want you to have this problem so they gave you the **DEG** and **RAD** statements. These statements may be used in a procedure. Or, they may be typed interactively while operating in the Debug mode. The syntax is simple:

```
D:DEG (RETURN)
D:RAD (RETURN)
```

The first command line sets a state flag in the system that tells the math routines to evaluate degrees. The latter sets a flag that says, "Hey I'm a radian".

Just in case you're curious and would like to see what a radian looks like, do this. Draw a circle and measure its radius. Then, mark off this length along the circle. Finally, draw an imaginary piece of pie by connecting these marks to the center of the circle. The angle you have just formed is a radian. Amazing!

FINDING THE STATE OF YOUR PROGRAM

After you've chased a bug in your code for a while, you'll probably lose yourself in a state of chaos. Consequently, you may need help because you can't see the forest for the trees. Rejoice, once again, BASIC09 can save the day. Type:

```
D:STATE (RETURN)
```

BASIC09 will respond with a status report. It should look something like this:

```
PROCEDURE DELTA
CALLED BY BETA
CALLED BY ALPHA
CALLED BY PROGRAM
```

This report tells you which part of the forest you are lost in. Seriously, it gives you a lot of information.

For example, it tells you that the program has stopped while executing a procedure named **DELTA**. This tells you, where you are. The procedure **DELTA** was called by a procedure named **BETA**. Now, you know how you got there. **BETA**, by the way, was called by **ALPHA**, which was called by your **PROGRAM**. By following each one of these limbs you should be able to find the problem.

LET ME CHANGE YOUR VALUE

Somehow that doesn't sound nice. Oh well, on with the lesson.

Did you ever play the "What If" game? What If 100,000 copies of this book are sold? I would probably faint but it sure is nice to dream. Or, what if we only sell five copies? Oh, well, Mom and Dad are sure to like it.

What's this leading to? Well, if you have a variable named **copies** in a procedure and you suspect something is rotten in the **PROCEDURE bookstore** you can use the debugger's **LET** statement to run some tests. These lines may help you get to the bottom of the plot:

```
D:LET copies := 5000
D:LET copies := 1000
D:LET copies := copies*2
```

In the first example, you are assigning the value 5000 to the variable **copies** interactively. After you have done this, you can type **CONT** to find the answer to your "What If" question.

The second debug command line sets the value of copies to 1000. Notice that while the = syntax is legal, the **:=** form is preferred because it clearly distinguishes an assignment operation from a comparison or test for equality.

The third command merely sets the value of **copies** to the value it had when the procedure was suspended multiplied by two.

There's only one hitch. When you use the **LET** statement in the debug mode, you must make sure that you use the same variable names that you used when you wrote the original source code. If you don't, you'll receive an error message and manage to get yourself more upset. Incidentally when working within the debugger, you can't use the **LET** statement to assign a new value to a user-defined data structure.

PRINTING THE CURRENT VALUE OF A VARIABLE

One of the quickest ways to spot a problem in your procedures is to take a quick look at the value of your variables at suspected problem points in your code. To do this, you use the **PRINT** statement. Again, the syntax is simple:

```
D:PRINT copies
```

The rules for using the **PRINT** statement are the same as those for the **LET** statement. You must use the same variable names in the **PRINT** command line that you used in your original source code. And, you can't **PRINT** a user-defined data structure. Nor, can you make up new variable names and expect to **PRINT** a value.

Here are two more simple procedures that you can use to exercise the debug mode. Have fun!

EXAMPLE NO. 6: PRINBI

```
PROCEDURE prinbi
   0000  REM by T.F. Ritter
   0012  REM prints the integer parameter value in binary
   0041  PARAM number:INTEGER
   0048  DIM position:INTEGER
   004F
   0050  FOR position=15 TO 0 STEP -1
   0066    IF number<0 THEN
   0072    PRINT "1";
   0078    ELSE PRINT "0";
   0081    ENDIF
   0083    number:=number+number
   008F  NEXT position
   009A  PRINT
   009C
   009D  END
```

Isn't it a pain to convert numbers from decimal to binary? Not any longer. Just **RUN** this procedure. The syntax is:

```
B:RUN PRINBI (32)
```

EXAMPLE NO. 7: DICESIM

```
PROCEDURE dicesim
   0000  DIM dice,points:INTEGER
   000B
   000C  FOR dice=1 TO 20
   001C    points:=INT(6*RND(1)+1)
   002F    PRINT "Dice Number:"; dice; "= "; points
   004E  NEXT dice
   0059  END
```

This procedure will show you how poor your chances are at Atlantic City or Las Vegas. You're time is better spent learning to program your computer with BASIC09.

SUMMARY

We hope your programs never overdose on bugs. But if they try, the tricks you've learned in this chapter should help keep your Mercedes on the road and the "pit stops" short.

You've learned how to force BASIC09 to enter the debug mode by putting a **PAUSE** statement in your procedures or typing a **CONTROL-C** to generate a keyboard interrupt while your program is running.

You've discovered how to turn on the debugger's trace mode with **TRON** and turn it of with **TROFF**.

You've learned how to examine the value of a variable with the **PRINT** statement and change it with a **LET** command.

And finally, you can now **CONT**inue program execution after making a change or **STEP** through your procedures a line or more at a time.

You should know how to operate BASIC09 pretty well by now. It's time to concentrate on unleashing the tremendous power of this programming language. We'll start by looking at its amazing data typing capability in Chapter Seven as we begin Part II of The Grand Tour.

CHAPTER SEVEN

TYPE variables

IT PAYS TO TYPE YOUR DATA

Welcome to The Official BASIC09 Tour Guide, Part II — where the fun begins. The previous chapters showed you how to use BASIC09. Now, we're ready to deal with the programming process.

DATA TYPE DEFINITION

There are many types of people in the world. Look closely, you'll see: tall people and short people; fat people and skinny people; young people and old people, etc.

Likewise there are many types of data that can be fed into a computer. BASIC09, for example, can ingest the data types **BYTE, INTEGER, REAL, BOOLEAN** and **STRING** the minute you bring it to life. We'll explain each of them in this chapter.

Experience tells you that you can't put 10 pounds of potatoes in a five pound bag. Likewise, you can't put a **REAL** number or a **STRING** of characters into a part of memory designed to hold an **INTEGER** number. They just won't fit.

The wise homemaker saves space in the kitchen by using the proper cabinet for each utensil. In the same manner, the smart programmer saves memory space when he uses the proper data type while assigning storage space for variables.

We'll introduce you to each of BASIC09's built-in data types in this chapter. Then, we'll show you how to combine them to define your own data types.

But first, if you really want to understand programming, you must take a brief moment to look at the machine. On the surface this miracle of modern electronics appears to be solving long, complex problems in a single breath. Actually, it's solving a very large number of extremely small problems — one after the other — until it arrives at the solution to the complex problem.

As a programmer, it's your job to figure out which small problems need to be solved to get the big answer. Once you know this, it's a simple matter to type a series of instructions that tell the computer how to solve these problems.

Of course, you must also insure that you tell the computer to solve these small problems in the proper order if you want a meaningful answer to your big problem. The process of organizing these instructions is called programming.

If you're going to tell your computer what to do, you had better tell it who or what to do it to. Your instructions must have information or data to operate on.

For example if you write a line of code to tell your computer to multiply two numbers, you must give it the two numbers before you can expect it to do anything. We call these numbers data.

Most of the time you will type your numbers on a keyboard. When you do this, you are inputting data. When you are done, the program will operate on it and print your result on an output device like your CRT terminal.

In the real world everything is categorized — or typed. Persons can be men or women. Students can be in high school or college. Servicemen can be officers or enlisted.

If you try to sell the customer an apple when he is starving for an orange, you'll go out of business. If you call a general a private, you'll wind up in the brig. Got the idea? You must always keep your "types" straight.

In the programming world, the consequences of mismatched data are just as severe. After all, how would you like to get a bill from the IRS computer taxing you for a net income of one million dollars during the past year when you actually made only one million cents? Stranger things have happened! We'll try to show you how to avoid these pitfalls in this chapter.

You'll learn about several data "types" that are built in to BASIC09. Programmers call them "atomic" data types. They include **BYTE**s, **INTEGERs, REAL** numbers, **STRINGs,** and **BOOLEAN** values.

Toward the end of the chapter, we'll even show you how to define your own data types. Before you're through, you'll know all about the following BASIC09 words.

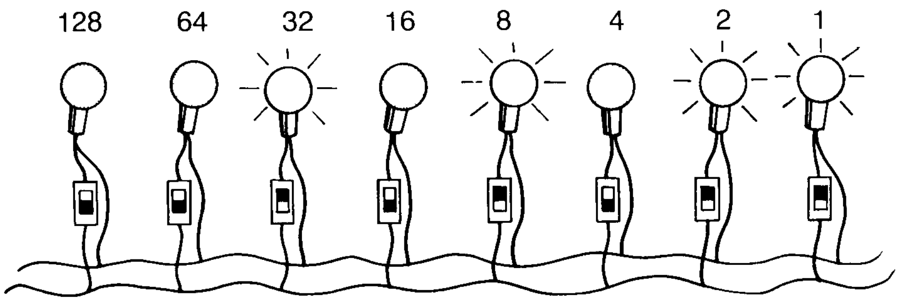
BYTE	**INTEGER**	**REAL**
STRING	**BOOLEAN**	**DIM**
TYPE		

BYTE ON

Before we bite off more than we can chew, let's digress for a moment and talk about how computers store our data.

Many personal and business computers today use a microprocessor that can work with eight bits of information at a time. Because of this fact, many memory chips are also designed to store eight bits of data at a time. A group of eight bits is called a "byte."

Let's picture a bit first. Imagine eight Christmas tree lights lined up side by side with a separate on/off switch under each one. Now, picture a piece of paper with a different number over each light. The numbers would appear in the following order:



Binary Value = 32 + 8 + 2 + 1 = 43

If a light is on, we will let it have the value equal to the number above it. If it is off, we will say that it has a value of zero. From here on out we will pretend that the words "light" and "bit" mean the same thing. Now, close your eyes and imagine that an elf comes along and turns off all the lights.

What are the eight bits worth? Not a dollar, that's for sure! If you add up the value of each of the eight lights — remember their value is zero when they are off — you'll find their total value is zero.

Pretend now that another elf comes by and turns on the light to the far left. What is the value of the eight lights now? You're right, they have a value of 128.

Now imagine that someone turns on the light on the far right. How much are the eight bits worth? Would you believe 129?

Yes, because 128, the value of the first light, plus one, the value of the light that was just turned on, is equal to 129. And the beat goes on. For every different combination of lights turned on, the total value of the eight bits is different. In fact these eight lights, can have a value ranging from zero when all the lights are off, to 255, when all lights are on.

If you can let your imagination carry you one more step, you'll get an "A" in the course. Try it. Imagine those eight light bulbs being eight memory cells in your computer. Each cell is a bit. The eight cells together form a byte. Just like the light bulbs — that byte can have a value ranging from zero to 255.

It may help you to think of a byte of memory as a little box where you can store a number. That box has two qualities — a name, and a value. The value can vary from zero to 255, depending upon what you put in it. We call the little box a variable since we can change its value at will.

BASIC09 uses only one location in memory to store a **BYTE** variable. For this reason, a number held in a **BYTE** variable must have a value between zero and 255. When you want your computer to store a variable as a **BYTE,** you must use a **DIM**ension statement. For example:

```
DIM age:BYTE
```

This statement causes BASIC09 to reserve one byte somewhere in memory. It also recognizes this byte of memory by the name you have just given it, "age". If later you write, "age = age + 1" the value stored in that byte will be increased by one.

Remember, this point. You should always use **DIM** statements to tell your computer how much memory you need to store a variable. Once you DIM a variable, the computer reserves just enough memory to hold that variable. And, when you use **DIM** statements, your computer does not use up unnecessary space for variables that is doesn't need for the program.

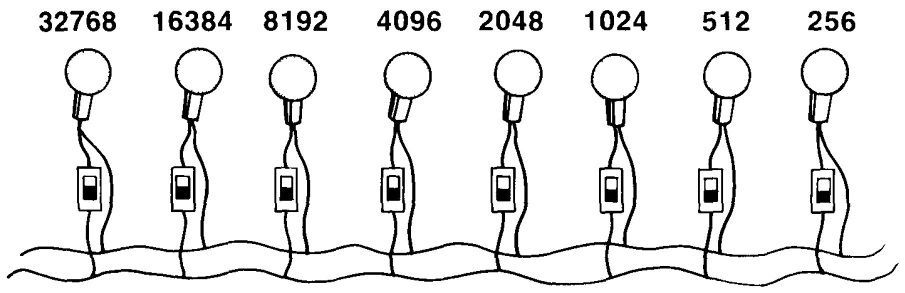
Here's something else you should know. When your computer uses the data stored in your **BYTE** variables, it converts them to **INTEGER** or **REAL** values before any calculations are made. This means that arithmetic performed on data of the type **BYTE** won't be any faster.

It still pays to **DIM**ension variables as type **BYTE** when you know for sure that all your data has a value between zero and 255, however. Why? Because it takes 16-bits — remember the light bulbs — or two locations in memory to store an **INTEGER** variable, and five locations to store a **REAL** number. A **BYTE** variable always takes one location.

You must be careful when you use **BYTE** and **INTEGER** variables together. If you attempt to store an **INTEGER** variable with a value greater than 255 in a variable of type **BYTE**, you could be in for a surprise. BASIC09 will only save the least significant eight bits of your 16-bit variable. Your program will come up with the wrong answer to the problem and you'll wonder why.

INTEGER VARIABLES ARE WORTH MORE

If the largest value one **BYTE** can store is 255, how large a number can you store in two bytes? Would you believe 65535? It's true. Let's string out an additional eight imaginary lights. The pieces of paper above them will read:



How can you recognize an INTEGER number? It's easy! You can tell by looking at the way it is printed on your terminal. INTEGERs are always printed without a decimal point.

Here is a sample list of legal INTEGER constants.

12	**-3000**	**64000**
$20	**$FFFE**	**$0**
0	**-12**	**-32768**

BASIC09 uses two locations in memory to store a variable of type **INTEGER.** The largest number that may be stored as an **INTEGER** is 65535. Yet, this is only true when the number is "unsigned" which is another way of saying it is always positive.

When you are using both positive and negative numbers — we call them signed **INTEGER**s — the 16th, or most significant bit, is used as a "sign" bit. If this bit is on — remember the light bulbs — the number stored is counted as negative. Conversely, if it is a zero, or off, the number is counted positive.

Since the 16th bit is now being used to indicate the sign of a number, it can no longer be added in as part of the value of the number. Since this bit position — or light bulb — has a value of 32768, this means that the largest positive integer number that can be stored in two memory locations is 32767. The largest negative number is -32768.

This means that you must be very careful when you are performing arithmetic or comparing integer numbers. Let's look at two addition operations.

```
TOTAL := 1000 + 1000
NEWTOTAL := 32767 + 1
```

After these assignment statements, **TOTAL** has a value of 2000. And, at first glance, you would expect **NEWTOTAL** to have a value of 32768.

Not true. **NEWTOTAL**'s value after this operation will actually be -32768. Since 32767 is the highest positive number that may be used in **INTEGER** arithmetic, BASIC09's math routines wrapped it around, modulo 65536.

The number 32768 has the 16th bit set. This means that it is a negative number. It's value is -32768. To help understand, let's look at some light bulbs again. This time a one means the bulb (bit) is on, a zero means it is off.

Positive Numbers:

```
     0:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     1:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
     2:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
     3:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
     4:  0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
      .
      .
      .
 32765:  0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
 32766:  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
 32767:  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Negative Numbers:

```
-32768:  1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-32767:  1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
-32766:  1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
-32765:  1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
-32764:  1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
      .
      .
      .
    -4:  1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
    -3:  1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
    -2:  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
    -1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

You must also be careful when you compare **INTEGER** type numbers between 32767 and 65535 because you are actually comparing negative numbers. For example, -32764 is greater than -32765 even though it's absolute value is smaller.

Because it is so easy to fall into this trap, you should get into the habit of checking only for equality, "=", or non-equality, "<>", when working with negative numbers. You should not check for greater than, ">", or less than, "<" conditions when working with these numbers. If you do, you'll only confuse the issue.

There's another thing you should know if you plan on dividing **INTEGER** numbers. The result will always have an **INTEGER** value. If there is a remainder, it will be discarded. Remember, **INTEGER** numbers never have anything to the right side of the decimal point. In fact, they won't even have a decimal point.

With these limitations why would you ever want to bother using them in the first place? There are two major reasons. The first is speed. Operations on **INTEGER** numbers are many times faster than operations on **REAL** numbers. The second is storage. It takes only two memory locations to hold an **INTEGER** number. It takes five to hold a **REAL** number.

INTEGER variables make very good loop counters. You can use a **REAL** number as an index in a loop. But, using an index of type **INTEGER** will speed up execution about three or four times.

INTEGER numbers are also ideal tools to use when you need to calculate addresses. They are especially handy in this application when you type them in hexadecimal form.

To do this, you type a dollar sign, "$" in front of the number. The lowest possible hexadecimal value of an **INTEGER** is zero, or $0000. The highest is $FFFF, the hexadecimal equivalent of 65535.

```
DIM DATE:INTEGER
```

Finally, if you want to insure that BASIC09 types a variable as an **INTEGER**, you must use the **DIM**ension statement shown above. The syntax is similar to that used to **DIM**ension a variable of type **BYTE.**

REAL NUMBERS

If you write a procedure and do not declare a numeric variable with a **DIM**ension statement, BASIC09 will automatically assume it is a **REAL** number. For the sake of readability however, it is a very good idea to always declare all variables.

A statement like this will do the trick for you.

DIM TWOPI:REAL

REAL numbers can be spotted easily — they're the ones with the decimal point. Sometimes they even sport an "E". More about that in a moment.

REAL numbers have a tremendous range. They can be as small as 2.938735877 times 10 raised to the -39th power. Written in decimal form, that would be:

.000000000000000000000000000000000000002938735877

Can you imagine counting all those zeros? It's no wonder BASIC09 prints it like this:

2.938735877E-39

By the way, the largest **REAL** number you can store is 1.701411835 times 10 raised to the 38th power. Here we go again:

170,141,183,500,000,000,000,000,000,000,000,000,000

Again, you could get dizzy counting the zeros.

For those who are technically inclined, BASIC09 uses five bytes to store each real number. The first byte holds the exponent of the number. The last four hold the mantissa. The sign of the number is stored in the least significant bit of the mantissa. Both the mantissa and exponent are in binary.

Here's a common number printed in the usual way — then printed in floating point notation. Let's look at the parts of a REAL number.

```
1000000000          1.0E+09
```

If you haven't bothered to count the zeros in the first line, there are nine. That's one billion.

In the second line, BASIC09 has printed the number in floating point notation. Let's look at it closer.

The mantissa — or value and sign of the number is 1.0. The exponent is E+09. Which means that the number will contain nine zeros if it is printed out.

The exponent can be either negative or positive. In this case it is positive, so the zeros go on the left side of the decimal point. If it were negative, the zeros would go to the right of the decimal point in front of the mantissa.

If you have not **DIM**ensioned a number to be of type **REAL** and you want BASIC09 to store it as a **REAL** number, you must type a decimal point when you enter it into your program. Again, BASIC09 will automatically print all **REAL** numbers with a decimal point.

Here are a few valid **REAL** numbers:

1.0	**9.8433218**	**-.01**
-999.000099	**100000000**	**5655.34532**
1.95E+12	**-99999.9E-33**	

If **REAL** numbers have a range like this, why would you ever want to use anything else?

There are two reasons. First, arithmetic operations with **REAL** numbers often take more than fives times as long as operations with **INTEGER** numbers. Secondly, **REAL** arithmetic may introduce rounding errors into your calculations.

Rounding errors occur because of the way your computer does arithmetic. For example, on some computers you could write an instruction that divides 1 by 3 which is then multipled by 3. What's the answer? Not 1, because the repeating fraction 1/3 cannot be represented exactly. Instead, the result must be represented as .33333333 which multiplied by 3 gives .999999999 instead of 1. But BASIC09 does careful rounding so it does come up with an answer of 1. Try it. Then try it on your friend's (non-BASIC09) computer.

Although the rounding effect has been minimized in BASIC09, you should still be aware of it. If you need absolute accuracy, you may want to use procedures to simulate long **INTEGER** operations.

If you are working with **REAL** numbers, you should be aware that it takes a lot less time to multiply a number than it does to divide. If you have a choice when you design your procedures, use multiply instead of divide when you can.

You must also be careful when comparing **REAL** numbers. Because of the rounding effect, two numbers that you think should be equal — may not be, exactly. This could cause a test to fail and send your procedure astray.

STRINGS HOLD A NUMBER OF CHARACTERS

STRINGs are groups of characters stored in consecutive locations in your computer's memory. They give you a place to store English language messages.

A **STRING** may be as short as one byte. If so, it would hold one character. Remember, it takes one byte of memory to store each character. On the other hand, the length of a **STRING** is limited only by the amount of memory available.

There are two ways to declare a **STRING** variable. You may use BASIC09's **DIM**ension statement. Or, you may append a $ to the variable's name. For the sake of better readability and more efficient memory usage, the **DIM**ension statement is the preferred method. Here are some examples.

```
Title$ := "The BASIC09 Tour Guide"
DIM Char:STRING[1]
DIM Page:STRING[1920]
DIM Answer:STRING
```

If you do not ask for a specific **STRING** length, BASIC09 will reserve 32 bytes of storage. For this reason, both Title$ and "Answer" may contain up to 32 characters. "Char" only has room to store one character, and "Page" may hold as many as 1920 characters.

These values are maximums only and do not represent the actual length of the string. For example, the length of Title$ after the assignment statement above is 22 characters. If you decide to change the title, you only need to change the assignment statement:

```
Title$ := "Microware's BASIC09"
```

You must however, make sure that the new name doesn't contain more than 32 characters. Just for the fun of it, try the following statements in a procedure.

```
Title$ := "BASIC09 — A Truely Loveable Language!"
PRINT Title$
```

Then run it and it will print:

```
BASIC09 — A Truely Loveable Lan
```

Did the message printed on your screen look line the line printed above? See what happens when you try to put 38 characters in a space that will only hold 32? BASIC09 does not print an error message when you make this mistake. It simply truncates your string. That's what happened above.

If you really needed Title$ to hold a STRING longer than 32 characters, you would have to explicitly declare it as a STRING using a **DIM**ension statement. You would also have to specify the size. This program line will do the job.

```
DIM Title:STRING[38]
```

What do you think would happen if you asked BASIC09 to "**PRINT** Title" immediately after you typed the line above? Standby for a surprise.

When BASIC09 executes the **DIM** statement above, it reserves 38 consecutive memory locations for you and gives them the name, "Title". It does not assign a value to them.

The characters that just happened to be stored in those 38 memory locations before BASIC09 executed your **DIM** statement are still there and you'll see them when you, "**PRINT** Title". Because of this, you must always assign your own value to your **STRING**s before you try to use them.

Incidentally, you may also create an empty **STRING.** This statement would do that:

Title := " "

After this statement is executed, "Title" could still hold 38 characters. But, it wouldn't.

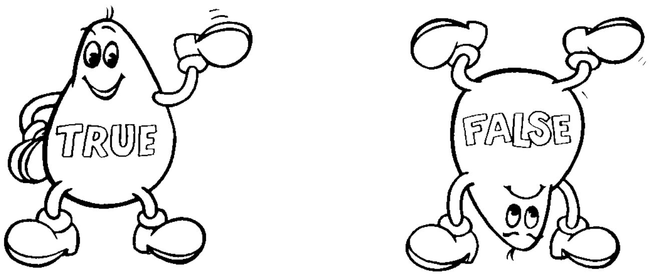
Rather, Title would have a value and a length equal to what you typed between the two quotation marks — absolutely nothing — or zero.

You have created a special case, the "null" **STRING.** It has a length of zero. If you print it, you will see nothing on your terminal. The concept is similar to setting an **INTEGER** or **REAL** variable to zero. It's sort of like drinking diet soda.

Sometimes — like when you're trying to find a small string inside a long string — you need to know the position of a character.

BASIC09 has functions that handle this task for you. But to use them, you must know where to start counting. In BASIC09, the first character of a string is always counted as character number one. This is always true — even when you are operating in the BASE 0 mode.

The BOOLEANS



BOOLEANS — THEY'RE EITHER TRUE OR FALSE

Sometimes things are so simple that they are very hard to comprehend. Such is often the case with data of the type **BOOLEAN.**

The range of values for **BOOLEAN** data is very short. It has only two. A **BOOLEAN** can be **TRUE.** Or, it can be **FALSE.**

BOOLEAN values can not be used for numeric calculations. But, they can be printed. If you print a **BOOLEAN,** you will see either a four or five letter word on your screen.

You will see the word, **TRUE** or the word **FALSE.** The one you see, depends on the value of your **BOOLEAN** data. I think we can clear this up by using an example.

```
PROCEDURE score
DIM won:BOOLEAN
DIM Our_Score,Their_Score:INTEGER
Our_Score := 12
Their_Score := 6
won := Our_Score > Their_Score
PRINT won
END
```

In the procedure above, we first declare "won" to be a variable of the type **BOOLEAN.** "Our_Score" and "Their_Score" are typed as **INTEGER.** Then, we assign a value to "Our_Score" and "Their_Score". The next line assigns a value to the variable "won". Here's how it shakes out.

Since Our_Score has been assigned a value of 12 and Their_Score has been set to a value of six, we are really asking BASIC09 to determine if 12 is greater than six. Because this evaluates as **TRUE,** the value of the variable "won" becomes **TRUE.** The final line in our procedure merely prints the value of "won" — or **TRUE** — on your terminal.

Here is another way to write the test for the **BOOLEAN** value of the variable "won'. Take your choice.

```
IF Our_Score > Their_Score THEN
won := TRUE
ENDIF
```

BOOLEAN variables are usually set to the result of an expression that makes a comparison. However, since they can also be set equal to the value of other **BOOLEAN** variables, they are often used as logical flags. You will often see them used with the **AND, OR, XOR,** and **NOT** operators. Let's try another example.

```
PROCEDURE Boolean_demo
DIM done,tired,quit:BOOLEAN

done := TRUE
tired := TRUE

quit := done AND tired

IF quit THEN
  PRINT "OK, You can Quit."
ELSE
  PRINT "Better Keep Going!"
ENDIF

END
```

In the **PROCEDURE** Boolean_demo, "done" **AND** "tired" are both set to a **BOOLEAN** value of **TRUE.** Thus, "quit" becomes **TRUE** in the assignment statement which follows. Therefore, when the **IF** statement is evaluated, BASIC09 will **PRINT** the message, "OK, You can Quit" on your terminal.

Be careful that you do not confuse the **BOOLEAN** operators above with the logical functions **LAND, LOR, LXOR,** and **LNOT.** The latter operate on **INTEGER** values bit by bit and will not work with variables of the type **BOOLEAN.**

BASIC09 does look after you here. If you try to store a non-**BOOLEAN** variable — for example a **REAL** or **INTEGER** number — in a variable of type **BOOLEAN,** you'll receive an error message when you try to run the procedure. And, if you're plotting to store a **BOOLEAN** value in a variable of another type, don't bother. You'll get the same "type mismatch" error.

AUTOMATIC TYPE CONVERSION

Here's a positive note. BASIC09 automatically converts all numeric data to the proper type during compilation. This conversion implies that you will seldom need to use the **FIX** or **FLOAT** operators. But, if you try to mix data types illegally, BASIC09 will report a "type mismatch" error when you exit the Edit mode.

ARRAYS CAN HOLD A LOT OF DATA

It's time to add some depth to our discussion of variables and data as we show you how to use arrays. Pretend that you want to put a message on your screen by printing the value of some **STRING** variables.

For example, let's say you want to print "I Program Well!". How would you do it? Here's the easy way.

```
PRINT "I Program Well"
```

Or, you could write:

```
DIM word:STRING[14]
word := "I Program Well"
PRINT word
```

But, if we let you do it the easy way, we couldn't show you the advantages of arrays. Let's divide that sentence up into three parts. Try this procedure.

```
PROCEDURE print_message1
DIM word1,word2,word3:STRING[15]

word1 := "I"
word2 := "Program"
word3 := "Well!"

PRINT
PRINT word1; " ";
PRINT word2; " ";
PRINT word3
PRINT

END
```

This procedure will work just fine. When you return to the system mode and type **RUN,** it will print, "I Program Well!" for you. Now, imagine that you want to print a sentence that has 15 words in it. You would be typing forever. There must be a better way.

Enter our friend, the array. Let's rewrite that procedure. Remember, we want it to print the message, "I Program Well!" Try this:

```
PROCEDURE print_message2
DIM words(15):STRING[15]
DIM wordcount,number_of_words:INTEGER

DATA "I", "Program","Well!"

number_of_words := 3

FOR wordcount := 1 TO number_of_words
  READ words(wordcount)
NEXT wordcount

PRINT

FOR wordcount := 1 TO number_of_words
  PRINT words(wordcount); " ";
NEXT wordcount
PRINT

END
```

The second procedure is the best alternative? Can you see its advantages?

The first procedure would take forever to write if you had more than a handful of words to print. The second can print any number of words just by changing the **DIM**ension of the array "words".

Besides, if you change the **READ** statement in the second procedure, you could easily pull the words you are printing from the keyboard or a disk file. By using a simple vector — another name for a one dimensional array — you have made your procedure many times more versatile.

You can also think of an array as a subscripted variable. Essentially, there are 15 variables named "words" — 15 little boxes if you will — in the procedure above.

Arrays are handy because they make it easy for you to assign a lot of values to a lot of variables. And since the subscript of a variable — wordcount in our procedure above — is also a variable, life is much easier.

There's one more thing you should understand about subscripted variables. The subscript is not the value assigned to the variable. For example, in our procedure above, when the value of wordcount is one, the value of words(wordcount) is "I". When wordcount has a value of two, words(wordcount) has a value of "Program", etc.

Here are two more statements that declare an array:

```
DIM screen(80,24):BYTE
DIM CoCoScreen(32,16):BYTE
DIM cube(3,3,3):BOOLEAN
```

You could use the first statement to store characters that you want to print on your standard 80 by 24 terminal. You could use the second to do the same thing on a TRS-80 Color Computer. The third could hold the status of each part of a Rubik's Cube.

In the first example — if you are operating in BASIC09's "**BASE 1**" mode — the character in the upper left hand corner of the screen will be named, "screen(1,1)".

If you ask for an array base of zero — by using the BASIC09 statement, "**BASE 0**" — this character will be "screen(0,0)". If you do not specifically ask for an array **BASE** of zero, BASIC09 will automatically assume that you want to use a **BASE** of one.

We showed you an array of strings because they are easy to read and understand. But, there is nothing stopping you from **DIM**ensioning an array of any BASIC09 data type. In fact, you can even build an array from complex data types that you define yourself.

COMPLEX DATA TYPES

What about the advanced programmer? How does he store a name, address and zip code in a computer's memory in an orderly fashion. Observe.

EXAMPLE NO. 4: GETNAME

```
PROCEDURE getname
 0000   (* Demo of complex data types *)
 0020   (* Input data into a complex name-address structure *)
 0056   (* Then print it to the terminal. *)
 007A
 007B   TYPE item=name,address(2):STRING[40]; zip:REAL
 009A
 009B   DIM record:item
 00A4
 00A5   PRINT
 00A7   PRINT "Please enter the data requested."
 00CB   PRINT "Type <RETURN> for a name to end this session."
 00FC
 00FD   LOOP
 00FF
 0100     PRINT
 0102     INPUT "Type a Person's Name: ",record.name
 0124
 0125   EXITIF record.name="" THEN
 0134   ENDEXIT
 0138
 0139     INPUT "First line of address: ",record.address(1)
 015E     INPUT "Second line of address: ",record.address(2)
 0184     INPUT "Type the person's ZIP code: ",record.zip
 01AC     RUN displayname(record)
 01B6
 01B7   ENDLOOP
 01BB
 01BC   END

PROCEDURE displayname
 0000   (* This procedure prints a record gathered by "GETNAME". *)
 003C
 003D   TYPE item=name,address(2):STRING[40]; zip:REAL
 005C
 005D   PARAM record:item
 0066
 0067   PRINT
 0069   PRINT "Here is your record: "
 0082   PRINT
 0084   PRINT record.name
 008C   PRINT record.address(1)
 0096   PRINT record.address(2);
 00A1   PRINT " ";
 00AB   PRINT record.zip
 00B3   PRINT
 00B5
 00B6   END
```

The procedure above demonstrates BASIC09's complex data types. The programmer has combined a **STRING,** an **ARRAY** of two **STRINGS,** and a **REAL** number into a singular data type called an item. He then declared that each record would look like an item in memory and proceeds to ask you to fill in the blanks.

When you type a name, it is stored in the variable called record.name, which is the first part of an item. Notice that you have 40 character spaces reserved in memory for your answer. It could just as easily have been set to 24 characters. It's all up to you, the programmer.

BASIC09's data typing features let you define variables to fit the shape of the data you must store. Additionally, they let you combine your own data types into larger data structures when they are needed. Further, they let you give English language names to data fields. Your programs are easier to read and understand.

The **PROCEDURE** "getname" also demonstrates the use of modules. In fact, it shows how entire data structures can be passed as parameters to other procedures. In the **PROCEDURE** "getname" an entire record is passed to the **PROCEDURE** "displayname". We'll talk more about modularity and parameters in Chapter 12. For now, we'll limit the discussion to data types.

What is a complex data type?

I'm glad you asked? The whole objective of a high level, interactive programming language like BASIC09 is to make your life easy. Complex Data Types do just that.

Pretend for a moment that you have been really successful in your programming efforts and you now have more than 500 regular customers. You've decided that it's about time to use your computer to make record keeping easier.

To do this, you need to write a procedure that will exercise a data file that holds each customer's name, address, and current balance. The first thing you must do is define your data structure. Here's one BASIC09 statement that will work.

```
TYPE customer_rec = name,address(2):STRING[40] ; balance:REAL
```

This statement tells BASIC09 that you want a customer record to be made up of four parts. The first part of the record holds your customer's name. It is a string variable named "name" and is **DIM**ensioned to hold 40 characters.

The second part of each record is an array that holds two elements. The first element in the array stores your customer's street address, the second holds his City, State, and zip code. The final part holds his current balance, a number of type **REAL.**

The statement before creates a new data "type" for you. It does not reserve any memory for your data however. To reserve memory you must use the **DIM**ension statement. And, since the **DIM** statement accepts user-defined types, you can now use customer__rec as a data type:

```
DIM customer__file(50):customer__rec
```

That's all it takes. You have defined an array and reserved space in memory for 50 customer records. Each one of those records can hold data of the type "customer__rec'.

To access the data in the array, you must use the field name as well as the array index. Let's look at a few sample assignments.

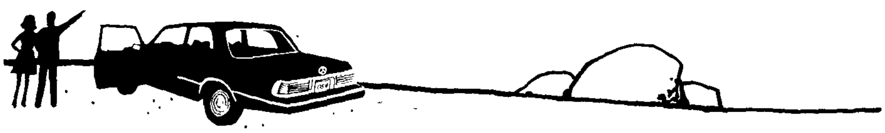
```
name$ := customer__file(10).name
Street$ := customer__file(10).address(1)
City__State$ := customer__file(10).address(2)
```

The first statement reads element number 10 (the 10th customer's name) from the array you named customer__file. It assigns the value it finds there to a **STRING** variable called, name$.

The second statement works the same, except it reads the first element in the array called "address', from the 10th element of an array called customer__file. It then assigns that value to the **STRING** variable named Street$. The third line reads the same customer's City, State and **ZIP** code from the second element in the array.

There's only one limit to your creativity. All new data types you define must be one dimensional.

However, since your data types can be made up of any of BASIC09's five atomic data types — as well as — any complex types that you have previously defined, there is almost no limit.



Here's a procedure that demonstrates a very complex data type.

EXAMPLE NUMBER 19: STRUCTST

```
PROCEDURE structure
     0000   (* example of intermixed array and record structures
     0034   (* note that structure d contains 200 real elements
     0067
     0068   TYPE a=one(2):REAL
     0078   TYPE b=two(10):a
     008A   TYPE c=three(10):b
     009C   DIM d,e:c
     00A9
     00AA   FOR i=1 TO 10
     00BC     FOR j=1 TO 10
     00CE       FOR k=1 TO 2
     00E0         PRINT d.three(i).two(j).one(k); " ";
     00FF         d.three(i).two(j).one(k):=3.1459
     0120         PRINT e.three(i).two(j).one(k); " ";
     013F       NEXT k
     014A     NEXT j
     0155   NEXT i
     0160
     0161   (* This is a complete structure assignment *)
     018E
     018F   e:=d
     0197   PRINT
     0199
     019A   FOR i=1 TO 10
     01AC     FOR j=1 TO 10
     01BE       FOR k=1 TO 2
     01D0         PRINT e.three(i).two(j).one(k); " ";
     01EF       NEXT k
     01FA     NEXT j
     0205   NEXT i
     0210
     0211   END
```

This procedure shows how you can mix arrays and record structures.

Remember, we tell BASIC09 that we need an array by entering the number of elements in parenthesis directly behind the name of a variable. In our procedure, the array is "one". Data of the type "a" contains one field which is an array, named "one", which contains two real numbers. BASIC09 is operating in its default **BASE1** mode.

How many elements would there be in the array "one" if we were operating in the **BASE0** mode? If you said three, your are right. In **BASE0** we would have element number zero, element number one, and element number two. In **BASE1** we have only element number one and element number two.

Now, let's look at data of the type "b". From the type statement we can determine that it is an array that holds 10 elements of type "a". Since each element of data type "a" holds two **REAL** numbers, data of type "b" will hold 20 **REAL** numbers. Got the idea?

When the pattern becomes clear, you'll wish you had invented the language. So, how about the data type named "c"? What does it hold? Let's look close.

Since data **TYPE** "c" holds 10 elements of **TYPE** "b", and "b" holds 20 **REAL** numbers, data of **TYPE** "c" contains 200 real numbers.

Now, look at the next line. Remember, the **DIM**ension statement is where you ask BASIC09 to reserve memory for your data. Here you are requesting enough memory to hold two complex variables — one named "d", the other named "e".

You are asking BASIC09 to reserve 2000 bytes of memory — 1000 for "d" and 1000 for "e". You need this much memory because both variables are of **TYPE** "c" and, therefore, contain 200 **REAL** numbers each. Did you remember that it takes five bytes of memory to store each **REAL** number?

Next, the procedure prints out the original value of each field of the complex variable named "d". It then sets each field in that variable equal to the value of "pi" and prints the value of each element of the complex variable "e".

The magic follows. The line "e: = d" sets complex variable "e" equal to complex variable "d". Then, to prove that it works, it reprints "e". Take a look for yourself. It really works.

Let's look at another example. Believe it or not, you may assign one type of data to a record of another type — if you know exactly what you are doing. Just be extremely careful when you do it. Study this procedure.

EXAMPLE NUMBER 20: CONVERT

```
PROCEDURE convert
     0000
     0001   TYPE simple = item:STRING[32]
     0011   TYPE complex = ascii(32):BYTE
     0021
     0022   DIM first:simple
     002B   DIM second:complex
     0034
     0035   PRINT "This procedure converts strings to decimal ASCII values."
     0071   INPUT "Enter a string less than 32 characters: ",first.item
     00A5
     00A6   second = first
     00AE
     00AF   FOR index = 1 TO LEN(first.item)
     00C6     PRINT second.ascii(index); " -";
     00D8   NEXT index
     00E3
     00E4   PRINT
     00E6
     00E7   END
```

This procedure shows you the quick and dirty way to convert a **STRING** of characters into an array of **BYTES.**

The first two lines define two new data **TYPEs.** Simple is defined as a **STRING** named item which can hold up to 32 characters and complex is defined as an array of 32 **BYTEs.**

The next two lines request memory for the variables first and second. Notice that first is **DIM**ensioned as **TYPE** simple, which means it contains a **STRING** with a field named item that can be up to 32 characters long. Second is defined to be of type complex, which means that it holds an array named ascii which holds 32 elements of type **BYTE.**

After requesting memory from BASIC09, the procedure asks you to type a **STRING.** Then, it sets the value of second equal to the value of first in one statement. Notice that this process works, even though the two variables are of different **TYPES.**

Do you see how you can get in trouble if you don't watch what you are doing. For example, what happens if you define complex to hold an array of **INTEGERs** instead of an array of **BYTES?**

The moral of the story. You can take short cuts but make sure you know exactly what you are doing.

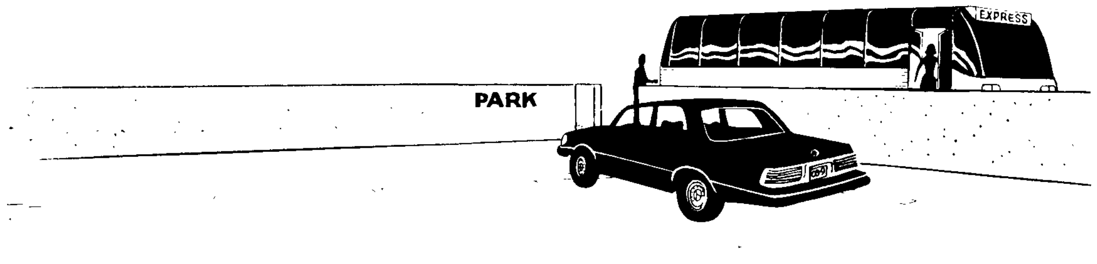
SUMMARY

It's about time for you to take a break. You've had your work cut out for you in this chapter. You've learned how data is stored in your computer and how it can take many different forms. You've been introduced to **BYTEs, INTEGERs, REALs, STRINGs, BOOLEANs,** and **ARRAYs.** You've even learned how to define your own data types.

Now that you know all about data, stand by! We'll introduce you to some real operators and expressions in the next chapter. You'll really begin to function as a programmer.

CHAPTER EIGHT

expressing yourself clearly



Welcome aboard the express bus. Don't worry, your Mercedes will be safe in the lighted parking lot while we take this short detour.

We're changing the approach a bit in this chapter. After introducing you to functions, operators, and expressions, we'll refer you to Part IV where you'll find an Encyclopedia of BASIC09 keywords.

In the encyclopedia you'll find a description of every reserved word in BASIC09 — presented in alphabetical order. Words are classified according to their function and sample procedures are listed. A "**RUN**" of each procedure is printed so that you can prove to yourself that it works. We'll also give you a list of related words that you may wish to study.

In this Chapter you'll be introduced to:

Assignment Statements
Control Statements
Directive Statements
Declarative Statements
Functions
Expression Operators

STATEMENTS — THEY DEFINE AN ACTION

You can compare a statement in a procedure to a sentence in a "how to" article. Statements tell your procedure what to do to your data. Most of the time, one statement tells your computer to perform only one task.

BASIC09 statements may contain up to 255 characters. Since most terminal's can only print 80 characters on a line, BASIC09 lets you use line feeds when you type long statements to divide your statement into two or more physical lines and makes it easier to read. Yet, the computer still sees only one statement.

You can also put more than one statement on a line in your programs by using a backslash character. "\", to separate individual statements. It is best to avoid this feature whenever possible, however, because it makes programs hard to read and edit.

BASIC09 procedures are made up of a series of statements. By completing a number of small tasks — one per statement — in the proper order, your computer can solve big problems.

Several types of statements are recognized by BASIC09. They each do a different job. Let's take a closer look.

ASSIGNMENT STATEMENTS

Assignment statements do just what their name implies — they assign a value to a variable or memory location.

In the last chapter we told you to think of the memory in your computer as a number of little boxes where you may store data. We also told you that that these boxes have a name and a value. Assignment statements change the value of the data in those little boxes. For example:

```
LET any_number := 10
LET double_number := any_number * 2
```

The colon and the equal sign together, ":=", are read as "becomes". Assignment statements can also use a single equal sign, "=" but this format is discouraged. Your programs will be easier to understand if you always use becomes, ":=", in an assignment statement and an equal sign, "=", when you are testing for equality.

After the two lines above are executed, the variable any_number has a value of 10 and double_number a value of 20.

Every time BASIC09 executes an assignment statement, the value of the variable named in the statement is changed. Any value that was stored there before is gone forever. Be careful — especially when you assign a value to the same variable more than once in the same program.

BASIC09 lets you take a shortcut when assigning values to variables. You do not have to use the word **LET.** Thus, the assignments above could be made like this:

```
any_number := 10
double_number := any_number * 2
```

CONTROL STATEMENTS

Control statements are like traffic cops. They control the flow of your procedures. We'll pick up the detail in Chapter Nine as we show you how to add structure to your programs. For now, here's a short sample:

```
PROCEDURE control_demo
DIM number:INTEGER

number := 1

WHILE number < 10 DO
  PRINT number
  number := number+1
ENDWHILE

END
```

The **WHILE** ... **DO** statement in the procedure above tells the computer to print the value of the variable "number" and then add one to it over and over again until the value reaches 10. When that happens BASIC09 jumps out of the loop and executes the statement on the line that follows the word **ENDWHILE.** Try it!

Directive Statements

Directive statements are used to tell BASIC09 or OS-9 to do something. Your jobs may range from changing your current data directory or current execution directory to turning the debugger's trace mode on and off. Here's a directive statement that tells BASIC09 to call the OS-9 Shell and print a listing of the directory on the disk you have installed in drive zero:

```
SHELL "dir e /d0"
```

Declarative Statements are used to define data **TYPEs** and request memory for data storage. We gave you the details in Chapter Seven.

Input/Output statements give you a way to talk to the outside world. Chapter 10 is dedicated to them.

And finally, operators are the verbs that tell your programming sentence which operation to perform.

FUNCTIONS PERFORM MANY JOBS

You can think of a function as a magic word that lets you perform some job automatically. For example, you use a function to find the square root of a number or the sine of an angle. When you type the name of the function, BASIC09 calls in a fast machine level subroutine to do your job. For example, here's how you find the square root of the number stored in variable "x".:

```
root := SQRT(x)
```

Basic09 has a vast assortment of different functions, so you'll find functions in the Encyclopedia to do many jobs. Some of them work with **INTEGER** and **REAL** numbers while others work with **STRING**s **of characters. A few perform logical operations and three return a BOOLEAN** value of **TRUE** or **FALSE.**

Many functions perform calculations on numbers that you supply in your statements. Most return a value of a specific data "type'. The Encyclopedia listings in Part IV show what you need to give a function and what you can expect from it. Sample procedures show many functions in action.

OPERATORS — THEY'RE ALL ACTIVE VERBS

Operators do the work in your procedures. With them you can add, subtract, multiply and, divide — and much more.

For example, if you want to tell your computer that a number should be negative, you simply type a minus sign, "-", in front of it. Here's how you do it.

```
LET a_number := -25
```

In our example, a_number has been negated. After this statement is run, the value of a_number is minus 25. When you run this statement, you are using BASIC09's "unary negation operator". How's that for a buzzword!

Another operator lets you raise a number to a power. Yet another allows you to join **STRING**s together with a plus sign, "+".

```
a_new_number := an_old_number ↑ 2
my_number := your_number ** 2
First_string$ := "Hello "
Second_string$ := "everyone!"
A_longer_string$ := First_string$ + Second_string$
```

The first two statements above run the same. You can type either the up arrow, "↑", or two asterisks, "**", to tell BASIC09 you want to raise a number to a power.

Another special group of operators lets you determine the relationship between the value of one variable or expression and another. The symbol for each operator and its English language meaning is listed here.

=	**equal to**
< >	**not equal to**
<	**less than**
≤	**less than or equal**
>	**greater than**
≥	**greater than or equal**

The symbols you see above are the actual operators you will use in your programs. The words merely describe them. Operators of this type return a Boolean value because they always evaluate as TRUE or FALSE.

Other Boolean operators are used when more than one condition needs to be tested at the same time. For this reason you'll often see **NOT, AND, OR** and **XOR** used with the **IF** ... **THEN** statement. For example:

```
IF 2 > 1 AND 3 < 4 THEN PRINT "You're Right!" \ ENDIF

IF 2 > 1 OR 4 < 3 THEN PRINT "Right Again!" \ ENDIF
```

When you must determine if one number or character is equal to another, you need some basis of comparison. BASIC09 uses the ASCII collating sequence. ASCII is an acronym that stands for the American Standard Code for Information Interchange.

Under the ASCII system each character has a unique numeric value, which is true for numbers, letters and any other character transmitted or received by your computer. Because of this standard our computers can communicate with terminals, printers, etc. They can also communicate with each other.

For example, the number zero, "0", in ASCII has a numeric value of 48 decimal; one, "1", has a value of 49; etc. If your computer sends mine a character with a decimal value of 48, my computer knows that yours is trying to send a zero, "0".

Here's a table that shows the ASCII value of capital letters in the alphabet.

LETTER	DECIMAL VALUE
A	65
B	66
C	67
D	68
.	.
.	.
.	.
X	88
Y	89
Z	90

OPERATOR PRECEDENCE

In society, some people have more power than others. Bankers get more done than bakers. Generals order sergeants to do almost anything.

Likewise, some operators have more power than others. Programmers call this precedence. Essentially it means that an expression is evaluated in a very specific order. According to BASIC09's rules of precedence, operations take place in the following order.

1. BASIC09 starts at the left side of an expression and moves towards the right.

2. Constants or variables that must be forced negative are handled.

3. Any power (exponentiation) operations are done.

4. At this point, BASIC09 starts over at the left end of the expression and does all multiplications and divisions.

5. Then, all additions and subtractions are done.

6. And finally, Boolean expressions are evaluated.

Sometimes, you need to override the natural precedence of the language. BASIC09 lets you do this by using parentheses. All expressions enclosed in parentheses are evaluated first. Remember this trick. It's a good thing to know. Let's look at an example.

```
6 + 2 * 4 * 3        (yields 30)
(6 + 2) * 4 * 3      (yields 96)
```

Notice that in the first example, multiplication is done first. In the second, the addition was done first since it was enclosed in parentheses.

If operators in an expression have equal precedence they are evaluated left to right. There is one exception however — exponentiation is evaluated right to left. Speaking of exponentiation, BASIC09 will not let you raise a negative number to a power.

EXPRESSIONS — THEY HAVE A VALUE

An expression is a set of rules used by a statement to tell a procedure how to compute a value. It combines one or more operands with operators. An operand may be a constant, the current value of a variable or the result returned by a function.

Expressions always evaluate to one of BASIC09's five "atomic" data types. Because of this, the result of an expression must be of "type" **BYTE, INTEGER, REAL, STRING** or **BOOLEAN.**

Data types may be mixed within expressions, in fact, many times the evaluation of an expression results in a value which has a different data type than its operands.

Here are some valid expressions:

```
a := b+c*2
a := (b+c+d)/e
a := b>c AND d>e OR c=e
a = b = c
```

The first example multiplies the current value of the variable "c" times two and adds the current value of the variable "b" to the result. The result is stored in the variable "a".

The second adds the current value of the variables "b", "c", and "d", and divides the result by the value of the variable "e". The result is put in the variable "a".

The third and fourth expressions may seem a little strange. In both, the variable "a" must be of type BOOLEAN because the expression on the right hand side of the "becomes" symbol, ":=", returns a **BOOLEAN** result.

The final expression evaluates as **TRUE** if the current value of the variables "a", "b" and, "c" is equal. Otherwise, it returns a value of FALSE. Again, the result is stored in the variable "a".

BASIC09 lets you mix the three numeric data types in an expression. Mixing is possible because BASIC09 does automatic type conversion before an operation.

AUTOMATIC TYPE CONVERSION

If you mix an **INTEGER** and a **REAL** number in an expression, the result will be of type **REAL** because **REAL** numbers have a greater range than **INTEGER** numbers.

You must be very careful when assigning the result of an expression to a variable of type **BYTE** or **INTEGER.** The result must always be within the range of the data **TYPE** you are assigning it to. If not, the result may be an error.

For example, when you assign an expression to a variable of data type **BYTE,** the result must have a value between zero and 255. Likewise, when you assign a result to a variable of type **INTEGER,** it must have a value between -32768 and +32767. If you feel like you need a review of BASIC09 data types, take time now to review Chapter Seven.

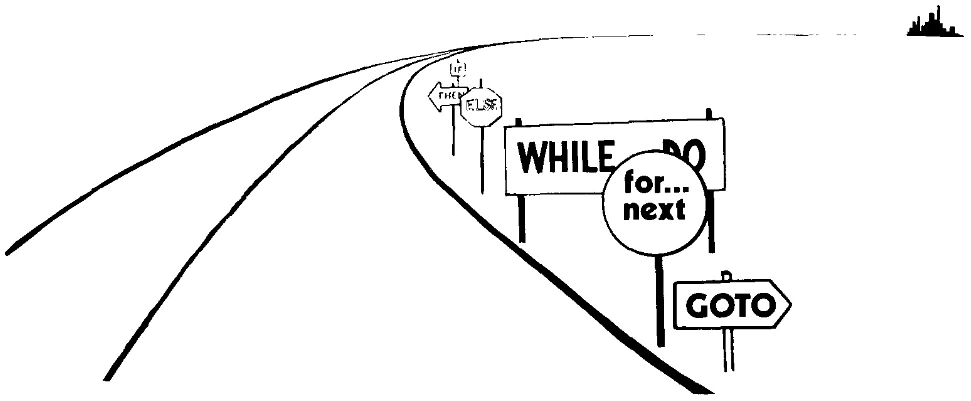
Remember, every statement, operator, and function of the language is described in our encyclopedia of BASIC09 in Part IV. Browse through it now and then use it as a reference while you program.

SUMMARY

In this chapter we gave you an overview of statements, operators, functions and expressions. Take a quick break now, and then join us in Chapter Nine where we'll show you how to add structure to your programs.

CHAPTER NINE

control structures let you go with the flow



A journalism professor once told me that life has no order. 'Writers must add structure to life to bring meaning to the reader', he said. What a challenge!

As a programmer, you face a similar challenge. You must add structure to your programs to bring meaning to the computer. If you fail to add structure, your programs may run around in circles and never do any work for you.

The real power of your computer lies in its ability to perform a number of simple tasks over and over again at great speed. It's up to you however, to tell it when to stop. If you don't — it won't. It will be perfectly content to sit there and do the same thing forever.

When you tell your computer to repeat a particular statement or series of statements many times, you are creating a "loop". Loops come in several different shapes and sizes. We'll introduce you to five in this chapter.

We'll show you how to force your computer to make a decision and how to trap expected errors without aborting your program. You'll be introduced to these control structures:

IF ... THEN ... ELSE
FOR ... NEXT
WHILE ... DO

```
REPEAT ... UNTIL
  LOOP ... ENDLOOP
  EXITIF ... THEN ... ENDEXIT

GOTO
  GOSUB
  ERROR
  ON ERROR GOTO
```

We'll start our discussion of structure by looking at two versions of the same procedure. The first example is written in standard BASIC and uses line numbers. The second takes advantage of BASIC09.

SIGNTEST

```
PROCEDURE oldsigntest
        0000 100 INPUT "Please type any number: ",x
        0023 110 IF x>0 THEN 150
        0036 120 IF x<0 THEN 170
        0049 130 PRINT "The number is zero."
        0063 140 GOTO 180
        006A 150 PRINT "The number is positive."
        0088 160 GOTO 180
        008F 170 PRINT "The number is negative."
        00AD 180 END
```

NEWSIGNTEST

```
PROCEDURE newsigntest
        0000
        0001      (* Show BASIC09's control structure *)
        0027      (* solving the Signtest problem *)
        0049
        004A      DIM number:INTEGER
        0051
        0052      INPUT  "Type a number: ",number
        0069
        006A      PRINT
        006C
        006D      IF number>0 THEN
        0079        PRINT "Your number is positive."
        0095      ELSE
        0099        IF number<0 THEN
        00A5          PRINT "Your number is negative."
        00C1        ELSE
        00C5          PRINT "Your number is zero."
        00DD        ENDIF
        00DF      ENDIF
        00E1
        00E2      PRINT
        00E4
        00E5      END
```

The PROCEDURE oldsigntest is simple, yet you can barely read it. If it were 80 lines long, you probably wouldn't even try. You just don't need the aggravation of meaningless line numbers.

Let's trace the flow of this procedure anyway. When you type RUN, the computer asks you to type a number. It compares the number you type to zero. If the number is greater than zero it goes to line 150 where it prints the message, "The number is positive." It then executes the instruction at line 160 which tells it to GOTO line 180 — the END of the program.

If you type a negative number, the test at line 110 fails and the program falls through to line 120. The result of the test on this line will be true and the program goes to line 170 where it prints the message, "The number is negative." Line 180 is executed next and the program ends.

If you type the numeral "0", the tests at line 110 and line 120 both fail, and the program falls through to line 130 and prints the message, "The number is zero." After printing the message, the program executes line 140 which tells it to go to line 180 — the end of the program.

As you followed the description of SIGNTEST's program flow, you probably had a few questions. Why use meaningless symbols like line numbers to control program flow? Why use meaningless names for variables? What on earth does "x" mean?

Now study the procedure newsigntest. Compare it with signtest. Which one is easier to understand? The IF ... THEN ... ELSE ... ENDIF construct is only one of the BASIC09 control structures you'll learn in this chapter.

OLDLOOPCOUNT

```
PROCEDURE oldloopcount
      0000
      0001       (* Show method of counting in standard BASIC *)
      0030
      0031       (* Note:  You must hit the "Control C" key *)
      005E       (* to get out of this loop *)
      007B
      007C 10    LET N=1
      0088 20    PRINT N
      0090 30    LET N=N+1
      00A0 40    GOTO 20
```

The procedure oldloopcount is written in standard BASIC. It uses line numbers and a meaningless single character for a variable name. It also uses the BASIC verb **GOTO** which tends to make your programs unstructured. Let's compare it to the procedure newloopcount.

NEWLOOPCOUNT

```
PROCEDURE loopcount
     0000
     0001      DIM loopindex,topcount:INTEGER
     000C
     000D      PRINT
     000F      INPUT "How high shall we count? ",topcount
     0030      PRINT
     0032
     0033      FOR loopindex=1 TO topcount
     0044        PRINT "Number "; loopindex
     0053      NEXT loopindex
     005E
     005F      PRINT
     0061      END
```

The procedure newloopcount uses two variables named, loopindex and topcount and declares them to be **INTEGER** numbers. It asks you how high you want to count and stores the value of the number you type in the variable topcount.

Another loop prints the message, "Number 1", etc. Each time the procedure runs through the loop, the number printed is increased by one because the statement "**NEXT** loopindex" increases its value by one. Finally, after printing the value of topcount, the procedure stops.

Can you tell the difference between standard BASIC and BASIC09? You can almost read the BASIC09 program in English. It documents itself. And it sure looks clean without those tacky line numbers.

Notice how the names of the variables loopindex and topcount tell you what they do. Notice also how all BASIC09 key words are automatically printed in uppercase letters to make them stand out from the variables which we typed using all lowercase letters. Actually, we left the keyboard in lower case mode and let BASIC09 do the work. Talk about lazy.

LOOP ... ENDLOOP — IT COULD GO ON FOREVER!

We'll start our detailed discussion of BASIC09 control structures with the most general loop — **LOOP** ... **ENDLOOP**.

Imagine that you want to write a program that prints a character on your terminal's screen every time you strike a key. Most terminals do this for you automatically, but a simple loop can also do the job.

```
PROCEDURE echo_forever
DIM char:BYTE
LOOP
  GET #0,char
  PUT #1,char
ENDLOOP
END
```

This procedure first gets a character from your keyboard and immediately sends it to your terminal's screen. Remember, the BASIC09 and the OS-9 operating system always use device number zero, "#0", as the standard input device and device number one, "#1", as the standard output device.

What happens after our procedure sends the first character to the screen? I'm glad you asked. When it hits the **ENDLOOP** statement, it goes back to the **LOOP** statement and starts over again. There is no elegant way to exit from this procedure. It will run forever — or at least until you hit the terminal with an axe or type "Control C" on the keyboard.

Are there other ways to escape? To find out, let's reach deeper into BASIC09's bag of tricks.

EXITIF — A WAY TO ESCAPE FROM A LOOP

EXITIF — here's a statement that sounds promising. Let's rewrite the procedure:

```
PROCEDURE echoforawhile

DIM char:BYTE

LOOP
  GET #0,char
EXITIF ASC(char)=$1B (* ESCAPE *)
  PRINT "Don't frown, you told me to ESCAPE."
ENDEXIT
  PUT #1,char
ENDLOOP
END
```

What do you think will happen when we run the procedure echoforawhile? Do you think it will echo characters until you type the **ASCII** <**ESCAPE**> key and then quit? Try it!

The word **EXITIF** gives you a way to escape from an endless loop. It is not limited to use with the **LOOP** ... **ENDLOOP** construct. You can also use it within a **FOR** ... **NEXT** loop, a **REPEAT** ... **UNTIL** loop, or a **WHILE** ... **ENDWHILE** loop, etc. Here's how it works.

When BASIC09 sees the word **EXITIF** it evaluates the **BOOLEAN** expression that follows it. If the **BOOLEAN** value is **FALSE,** the procedure goes on to the line following the word **ENDEXIT** and continues.

If however, the statement following **EXITIF** evaluates as **TRUE,** your procedure executes the instructions between the word **EXITIF** and the word **ENDEXIT.** It then continues with the statement following the word **ENDLOOP.**

EXITIF is a very handy word since it lets you take care of any unfinished business when you exit a loop. It's always a good idea to clean up your mess before you move on.

A programmers life is much simpler when he uses **EXITIF** to trap errors and exit from an illegal situation gracefully. It's even better when he lets **EXITIF** print a short message to explain what happened at the same time. Try this!

```
PROCEDURE exitgracefully
DIM num,val,minimum:INTEGER

num := 100\ minimum := -10

REPEAT
  num := num-1
EXITIF num <0 THEN
  PRINT "Error — You can't take the square";
  PRINT "root of a negative number!"
ENDEXIT
  val := val + SQRT(num)
UNTIL num < minimum
END
```

What happened when you ran this procedure? Did you notice that the value of the variable "num" never comes close to the value of "minimum'?

We assigned -10 as the value of minimum early in the procedure, but we can never arrive there because we told the **EXITIF** statement not to let the value of the variable num go below zero. Essentially, we have let the **EXITIF** statement catch an error in our program. Then, we let it print a message to tell us what happened.

CONTROL OF PROGRAM FLOW

In standard **BASIC** the programmer's control of program flow is limited. Most versions allow only the "**GOTO**", "**GOSUB**", "**IF-THEN-<LINE NUMBER>** ", and "**FOR-NEXT**" constructs.

BASIC09 gives you several additional looping constructs so you may pick the best loop for the job. You can use a **WHILE** ... **DO** loop or a **REPEAT** ... **UNTIL** loop. Or, you can pick a structure that loops forever. It's called **LOOP** ... **ENDLOOP.**

Here's another example where the programmer escapes from a BASIC09 loop in a special case by using the **EXITIF** statement. Enter and **RUN** the procedure multiply.

MULTIPLY

```
PROCEDURE multiply
        0000
        0001     REM Demo of EXITIF-ENDEXIT and LOOP-ENDLOOP
        002B     REM Multiplies two real numbers for user and prints result
        0064
        0065     DIM multiplier,multiplicand,product:REAL
        0074
        0075     PRINT
        0077     PRINT
        0079     PRINT "Type the two numbers you would like to multiply."
        00AD     PRINT "Separate them by commas.  Example: 6,3 <return> "
        00E1     PRINT
        00E3     PRINT "Type two zero's to quit. "
        0100     PRINT
        0102     PRINT "Enter your numbers here: "
        011F     PRINT
        0121     PRINT
        0123
        0124     LOOP
        0126
        0127       INPUT multiplicand,multiplier
        0130
        0131     EXITIF multiplier=0 THEN
        013E       PRINT
        0140       PRINT "It was nice working for you!"
        0160       PRINT "Bye now."
        016C       PRINT
        016E     ENDEXIT
        0172
        0173       product=multiplicand*multiplier
        017F
        0180       PRINT multiplicand; " times "; multiplier; " equals "; product
        01A2       PRINT
        01A4
        01A5     ENDLOOP
        01A9
        01AA     END
        01AC
```

When you run the procedure "multiply", you are asked to enter two numbers. The procedure then multiplies the two numbers and prints the result. It will continue then until you tell it you want to multiply a number by zero. When you type a "0", you trigger the **EXITIF** statement, a thank you message is printed and the program ends.

Notice that BASIC09 automatically indents statements within a loop, giving you a visual image of program flow that becomes very important when your programs get longer and more complicated.

REPEAT ... UNTIL YOU GET TIRED

We've already given you a sneak preview of the words **REPEAT** ... **UNTIL.** Now, we'll give you the details.

REPEAT ... **UNTIL** is just another loop. It differs from **LOOP** ... **ENDLOOP** because it contains a built-in test that tells it when to stop. It makes this test at the bottom of each pass. If the expression tested evaluates as **TRUE,** the loop is exited.

Here's the important thing to remember about a **REPEAT** ... **UNTIL** loop. It always executes the statements within the loop at least once because the test always takes place at the bottom of the loop.

Let's study an example.

```
PROCEDURE factorial1
     0000      (* Compute Factorial:  number!  *)
     0022
     0023      DIM temp,number:REAL
     002E
     002F      temp:=1
     0037      INPUT "What number would you like the factorial of? ",number
     006C
     006D      REPEAT
     006F        temp:=temp*number
     007B        number:=number-1
     0087      UNTIL number<=1
     0093      PRINT "The factorial is "; temp
     00AC
     00AD      END
```

The procedure 'factorial1' computes the factorial of a number. It uses a **REPEAT** ... **UNTIL** loop to multiply a temporary number by the value stored in the variable "number". Each time it goes through the loop it sets a temporary number equal to itself times "number". Then, it subtracts one from the value of "number".

When "number" becomes less than or equal to one, the loop is exited and the value of the number is printed. This number is the factorial of the number you typed.

You can also compute the value of a factorial of a number with a **FOR** ... **NEXT** loop. We'll try that next.

LET'S LOOP FOR AWHILE NEXT WEEK!

The **FOR** ... **NEXT** loop executes a series of statements an exact number of times. Study the procedure factorial.

FACTORIAL

```
PROCEDURE factorial
      0000
      0001      PARAM number:INTEGER
      0008
      0009      DIM counter:INTEGER
      0010      DIM factorial:REAL
      0017
      0018      factorial: = 1
      0020
      0021      FOR counter = 1 TO number
      0032        factorial: = factorial*counter
      003F      NEXT counter
      004A
      004B      PRINT counter-1,factorial
      0057
      0058      END
```

When you run the procedure factorial, you must tell BASIC09 the number you want to compute the factorial of. You do this when you type the **RUN** command. For example:

```
RUN factorial(5)
```

This line, typed as a command line while in BASIC09's system mode, or entered as a statement in another procedure, tells BASIC09 to compute the factorial of the number five and print it on your terminal.

The number you type in parentheses is a parameter. You'll learn about parameters in Chapter 12. For now, we'll concentrate on the **FOR** ... **NEXT** loop.

Each time the procedure factorial runs through the loop, the line between the FOR and NEXT statements is executed. When the counter equals five — the value typed in the RUN command — looping stops and the procedure executes the line following the **NEXT** statement where it prints the result.

In the procedure factorial, BASIC09 automatically increases the loop counter by one each time through the loop. It's also possible to select the size of the counter's increase by using the word **STEP.** For example, if you want to count nickels:

```
PROCEDURE countbyfive
DIM cents,nickel:INTEGER
FOR nickel := 1 to 20 STEP 5
  PRINT "nickel number "; nickel
  PRINT nickel*5; " cents"
NEXT nickel

PRINT "We now have a dollar!"
END
```

What happens when you need to count backwards? And maybe you would like to ring the bell on your terminal when you go broke.

Let's rewrite the procedure.

```
PROCEDURE broke
     0000
     0001      DIM cents,nickel:INTEGER
     000C
     000D      bell:=7
     0015
     0016      FOR nickel:=20 TO 0 STEP -5
     002C        PRINT nickel; " nickels: ";
     003F        PRINT nickel*5; " cents"
     0050      NEXT nickel
     005B
     005C      PRINT CHR$(bell)
     0063      PRINT "Whoops!  We're broke!"
      007C
     007D      END
```

Use the two procedures above to study the **FOR** ... **NEXT** construct. Change the procedures to make them do different things within the loop. Then, change the amount the loop count increases or decreases each time through the loop by altering the **STEP** value.

When you feel like you have **FOR** ... **NEXT** loops under control, take another big step. Rewrite one of the procedures and try to do the same thing using a **REPEAT** ... **UNTIL** loop construct. Be brave.

WHILE YOU'RE STILL LEARNING, LET'S DO IT AGAIN

The **WHILE** ... **DO** ... **ENDWHILE** loop is another powerful BASIC09 construct. It's similar to the **REPEAT** ... **UNTIL** loop we studied earlier — with one significant change.

The **WHILE** ... **DO** loop always makes its test at the top of the loop. It's possible that your loop may never be executed. Consider this procedure:

```
PROCEDURE shallweprint
firstnumber := 100
secondnumber := 10
WHILE firstnumber < secondnumber DO
  PRINT firstnumber
  firstnumber := firstnumber +1
ENDWHILE

END
```

If you run this procedure, nothing will be printed on your terminal. Do you understand why? Let's make a quick pass through the procedure.

In the first two lines, the variable "firstnumber" is assigned a value of 100, and "secondnumber" is assigned to a value of 10. Next, the **WHILE** ... **DO** statement asks BASIC09 to evaluate the expression "firstnumber x secondnumber".

It is really evaluating "100 × 10". Since this expression has a **BOOLEAN** value of **FALSE,** BASIC09 never executes the loop. Rather, it immediately skips to the line following the **ENDWHILE** statement which is the **END** of the procedure.

Here's another procedure that uses the **WHILE** ... **DO** loop. It uses a variable of type **BOOLEAN** to determine if your cursor has passed column number 50. When it does, the value of "yes" becomes **FALSE** and the loop is exited.

```
PROCEDURE positiontest
DIM yes:BOOLEAN
yes := TRUE
WHILE yes DO
  PRINT "OK ";
  yes := POS × 50
ENDWHILE
END
```

Here's a tip. If you really want to understand what is going on inside a loop, add a **BREAK** statement in the first line of these short example procedures. Then, use the debugger to **STEP** through the procedure with the **TRACE ON.**

Here's one more for the road — a short procedure that reverses the characters in a word using a **WHILE** ... **DO** loop.

```
PROCEDURE reverseletters

DIM backward,word:STRING

backward := ""
INPUT word
```

```
WHILE LEN(word) > 0 DO
  backward := backward + RIGHT$(word,1)
  word := LEFT$(word,LEN(word)-1)
ENDWHILE

word := backward
PRINT word
END
```

IF YOU CAN THINK, THEN MAKE A DECISION

You can let BASIC09 decide the flow of a procedure by using the **IF** ... **THEN** ... **ENDIF** statement. **IF** the condition you are testing is **TRUE,** the procedure executes one set of statements — **IF** it is **FALSE,** it executes another.

Whenever you see an **IF,** you'll also see one of the **BOOLEAN** comparison operators we introduced in Chapter Eight. If you aren't familiar with these symbols and what they mean, take the time now for a quick review.

We'll show you a sample program line here for the sake of completeness. Then, I hope you forget that you ever saw it.

IF mynumber < yournumber **THEN** 200

Read in English, it means, "if the expression 'mynumber < yournumber' is true then **GOTO** line 200 and execute it." This example is the classic form of the **IF** ... **THEN** construct and in older BASICs it was almost the only form you ever saw.

Fortunately, BASIC09 is highly structured and does not make you use line numbers. In fact, you should avoid using them whenever possible. They just take up memory and slow down your program's execution. Let's look at the power of BASIC09's **IF** ... **THEN** ... **ELSE** ... **ENDIF** statements now.

POWERS

```
PROCEDURE powers
     0000
     0001     (* Procedure Demonstrates nexted "If-Then-(Else)-Endif" construct *)
     0045     (* Takes input value to given power *)
     006B
     006C     DIM value,result:REAL
     0077     DIM power:INTEGER
     007E
     007F     PRINT
     0081     PRINT "This program prints the powers of real numbers."
     00B4     PRINT "Maximum power=4; Type a '0' for power to quit."
     00E6
```

```
00E7     LOOP
00E9
00EA       PRINT
00EC       INPUT "Type a number here: ",value
0108       INPUT "Which power would you like to see? ",power
0133       PRINT
0135
0136     EXITIF power=0 THEN
0142       PRINT "Nice working for you — bye!"
0162       PRINT
0164     ENDEXIT
0168
0169       IF power=1 THEN
0175         result=value
017D       ELSE
0181         IF power=2 THEN
018D           result=value*value
0199         ELSE
019D           IF power=3 THEN
01A9             result=value*value*value
01B9           ELSE
01BD             IF power=4 THEN
01C9               result=value*value*value*value
01DD             ELSE
01E1               PRINT "Illegal value!!!"
01F5               result=0
01FD             ENDIF
01FF           ENDIF
0201         ENDIF
0203       ENDIF
0205
0206       PRINT result
020B
020C     ENDLOOP
0210
0211     END
```

The procedure, powers, illustrates BASIC09's powerful **IF** ... **THEN** ... **ELSE** ... **ENDIF** construct. The programmer's logic is clear because of the indented listing. But more importantly, this structure lets you write most programs without line numbers.

For those of you who have programmed before; yes there is a way to do this without all the **IF** ... **THEN** statements, but this procedure demonstrates the nesting of **IF** ... **THEN** statements and shows how BASIC09 automatically indents your program lines to enhance readability.

Since **IF** statements can be nested to any depth, extremely complex choices can be made. The procedure powers can make a five-way branch. The flow is determined by your answer to the prompt, "Which power would you like to see?" Even more complicated choices and decisions are possible. The only limit is the programmer's imagination. Here's another procedure that uses the **IF** ... **THEN** ... **ELSE** ... **ENDIF** construct.

```
PROCEDURE realtionship__test
    0000
    0001        DIM number,another__number:INTEGER
    000C
    000D        number: = 6
    0014        another__number: = 8
    001B
    001C        IF number<another__number THEN
    0029          PRINT "Number is less than another__number."
    0050        ELSE
    0054
    0055          IF number=another__number THEN
    0062            PRINT "Number equals another__number."
    0083          ELSE
    0087            PRINT "Number is greater than another__number."
    00B1          ENDIF
    00B3        ENDIF
    00B5
    00B6        END
```

Let's follow the execution of the procedure Relationship__test. First, the **BOOLEAN** expression "number x another__number" is evaluated. Since the result is **TRUE,** BASIC09 immediately executes the statement between the word **THEN** and the word **ELSE** and prints the message, "Number is less than another__number."

Remember, when the word **ELSE** is not used, all statements between the word **THEN** and the word **ENDIF** are executed.

Had the **BOOLEAN** expression above evaluated as **FALSE,** the procedure flow would have followed a different course. If the variable number had been assigned a value equal to the value of another__number, BASIC09 would have printed the message, "Number is equal to another__number."

Run this procedure several times using different values for "number" and "another__number". Remember to add the word **BREAK** so that you can use BASIC09's debugger with the **TRACE ON** to see what is going on.

GOSUB CALLS A BASIC09 SUBROUTINE

Subroutines can be very handy and save you a lot of typing.

A subroutine is an isolated part of your program which performs a specific function and then returns control to the main body of your program. In BASIC09, you execute a subroutine with the word **GOSUB.** To return to the main body of your procedure, you use the word **RETURN.**

There's a catch though. You must put a line number in front of the first line of your subroutine. In this case you must violate our informal rule that told you not to use line numbers. We'll let you get by with it this time!

Here's a short procedure that uses a subroutine to divide a number by two and print the result. The main body of the procedure calls the subroutine 20 times — 10 times to print half of the sine of a number and 10 to print half the cosine of another number.

```
PROCEDURE useagosub
     0000
     0001      DIM result:REAL
     0008      DIM number:INTEGER
     000F
     0010      FOR number:=1 TO 10
     0020        result:=SIN(number)
     002A        GOSUB 100
     002E      NEXT number
     0039
     003A      FOR number:=1 TO 10
     004A        result:=COS(number)
     0054        GOSUB 100
     0058      NEXT number
     0063
     0064      STOP
     0066
     0067      100 result:=result/2
     0076      PRINT result
     007B      RETURN
     007D
     007E      END
```

This procedure will give you another chance to use BASIC09's debugger and its tracing ability to study the flow of a procedure.

ON ... **GOSUB:** When You Need More Variety

Another form of the **GOSUB** statement evaluates an expression and uses the result to determine the line number of your subroutine. It takes the form:

ON xray **GOSUB 1000,2000,3000**

In this case xray must be of the type **INTEGER** and have a value between 1 and 3. If xray evaluates as 1, the procedure calls the subroutine located at line 1000. If it evaluates as 2, the procedure calls a subroutine at line 2000. And finally, if it is 3, the subroutine at line 3000 is called.

There is one additional rule you should follow when using this construct. The value of xray should never be greater than the number of line numbers supplied in the list following the word **GOSUB.** For example, if you somehow make a mistake and a value of 4 is assigned to xray, your procedure will not call a subroutine. Rather, it will execute the next statement in your procedure.

BASIC09's **ON** ... **GOSUB** structure is similar to that of the Pascal word **CASE** and is very handy in certain situations.

Here's an example that shows you how to deal with the seven days in a week. Imagine that you have written a procedure to compute the day of the week. Your procedure probably will give you a result that is a number. For example, Sunday would be day number 1; Monday, number 2 ... Saturday, number 7.

Here's the problem. People think of the days in a week in terms of Sunday ... Saturday, not in terms of 1 ... 7. How can you humanize your report.

You could use the classic **IF** ... **THEN** ... **ELSE** structure that we showed you earlier to print an English language message. Unfortunately, the procedure would become extremely cumbersome after two or three days. Let's look!

```
IF DAY=1 THEN
  PRINT "Sunday"
ELSE
  IF DAY=2 THEN
    PRINT "Monday"
  ELSE
    IF DAY=3 THEN
      PRINT "Tuesday"
    ELSE
(* Etc. *)
```

Imagine what this procedure would look like by the time you got around to Saturday. Granted, BASIC09's pretty-printing will make it easier to read but let's look for an alternative.

In **PASCAL** the procedure above would look like this:

```
CASE day OF
  1: WRITE ("Sunday");
  2: WRITE ("Monday");
  3: WRITE ("Tuesday");
(* Etc. *)
```

This procedure is much easier to read and understand. Now here's the good news. With BASIC09's **ON** ... **GOSUB** control structure you can do the same thing.

```
PROCEDURE dayofweek
  DIM day:INTEGER
```

```
    day := 1+INT(RND(6))
    ON day GOSUB 10,20,30,40,50,60,70
    END

10 PRINT "Sunday" / RETURN
20 PRINT "Monday" / RETURN
30 PRINT "Tuesday" / RETURN
40 PRINT "Wednesday" / RETURN
50 PRINT "Thursday" / RETURN
60 PRINT "Friday" / RETURN
70 PRINT "Saturday" / RETURN
```

ON ... GOTO is a similar control structure. The difference between the two shows up after the line is executed.

In the case of **ON** ... **GOSUB,** control always returns to the line following the construct.

When an **ON** ... **GOTO** statement is executed, control is transferred to the appropriate line number forever. It never **RETURNS.** Since serious problems can be created, good programmers avoid any use of the verb **GOTO** unless they have a very good reason.

GOTO: USE IT SPARINGLY

There comes a time in the life of every programmer when he must use a **GOTO** statement. Perhaps the flow of control needed to solve the problem is too irregular to be expressed in terms of the other control structures.

Just remember, the **GOTO** statement unconditionally transfers the control of your program to the line having the specified number. By the way, in BASIC09 the line number must be a constant. You may not use a variable or expression to represent it.

One of the most important criterion in programming style is whether or not the program can be understood by other readers. This criterion means that even though **GOTO** statements should be avoided, they are acceptable in exceptional cases where they let you avoid an awkward expression that is hard to understand.

One place where you may need to use a **GOTO** statement is in a procedure that must read data into a complex data type. You must always make sure that your procedure checks data and issues an appropriate error message to the operator.

But, what happens when you detect a data entry error that would totally confuse later procedures and cause erroneous answers?

The answer is simple — use a **GOTO** statement to jump out of the control loop your procedure is executing when the error occurs.

ON ERROR GOTO LETS YOU EXIT GRACEFULLY

Sooner or later, it will happen to you. You'll make a mistake when you define your algorithm and your procedure will generate an error. What then?

Normally, BASIC09 terminates your procedure and enters the **DEBUG** mode, which is fine for testing, but your face will sure be red when it happens to someone else that is trying to run your program. What can you do to avoid this potentially embarrassing situation?

Enter our latest hero — the **ON ERROR GOTO** statement. This statement "traps" any errors that occur and doesn't let BASIC09 enter the debug mode. Instead, it transfers control to a line number containing an error handling routine.

When you write your error handling routine, you'll most likely discover yourself using the **ERR** function. This BASIC09 word returns a number that represents a specific error. The number can be looked up in the BASIC09 reference manual where an English language description of each error is listed by number.

To use the function **ERR,** you assign its value to a variable. You must do this because **ERR** resets itself to a value of zero after it is called. Here's a short example:

```
PROCEDURE traperrordemo

DIM path,errnum: INTEGER
DIM name: STRING[45]
DIM line: STRING[80]

ON ERROR GOTO 10

INPUT "Filename? "; name
OPEN #path, line
LOOP
  READ #path, line
  PRINT line
ENDLOOP

  10 errnum := ERR
IF errnum = 211 THEN
  (* end of file error *)
  PRINT "Listing Complete."
  CLOSE #path
ELSE (* report other errors *)
  PRINT "Error Number "; errnum
ENDIF
END
```

The main procedure above simply opens the file that you select and lists it on your terminal, a line at a time. It does this until an error occurs.

Since you always receive an "end-of-file" error if you try to read past the end of a file, you planned ahead to trap this error with the routine beginning at line 10.

Line 10 checks to see if the error generated actually was an end-of-file error. If it was, it closes the file and exits the procedure. If not, it reports the error and exits.

Let's drop one more word on you here. Then, we'll exit this chapter. Sometimes it is handy to be able to generate errors on purpose. How else are you going to check your error handling routines.

To check error handling you may use the BASIC09 word, **ERROR.** This statement will generate an error with the error code that you specify. It looks like this:

```
ERROR(211) (* generates end of file error *)
ERROR(x-y) (* generates error code equal to the *)
(* value of the expression "x-y" *)
```

SUMMARY

You should feel like you have control of BASIC09 by now. In this chapter, you have been introduced to a group of constructs that give you control of your program's structure.

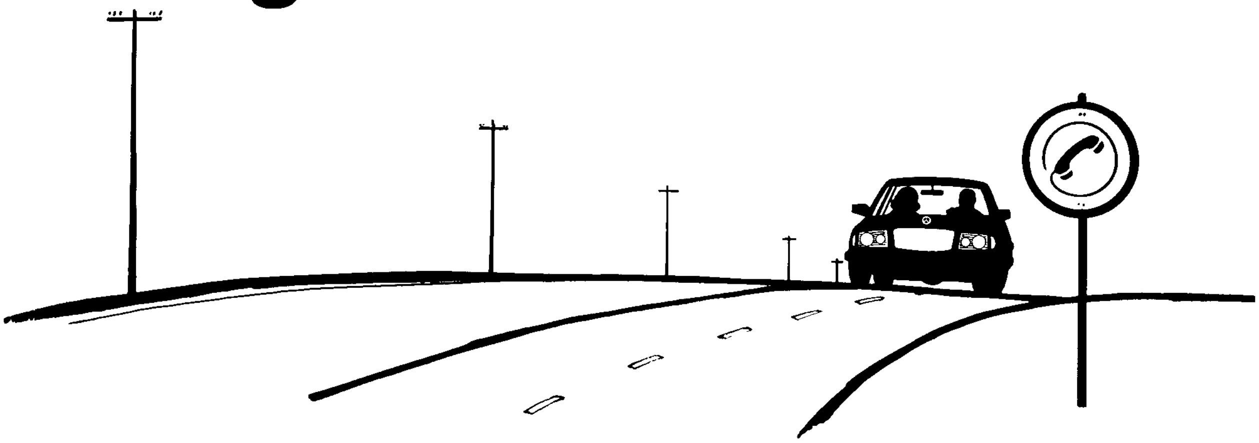
By controlling structure, you control the flow of your program and come closer to guaranteeing that you will compute the correct answer, which should be your ultimate goal.

By controlling your program's structure, you are also making your program easy to read and understand. A complete stranger should be able to understand it after a quick reading. Besides, you may be called in to make a change a year or two after you write a program. What are you going to do if you can't read your own code? Remember, forewarned is forearmed.

In Chapter 10 we'll move on down the road and show how your programs can communicate with the rest of the world. We'll introduce you to files and show you how to store data on semi-permanent magnetic storage devices. Your data will no longer disappear when you turn off the computer.

CHAPTER TEN

talking to the outside world



So far, we've only worked with data stored in your computer's memory or data sent and received from your terminal. We've shown you how to define data types and reserve memory space. We've shown how you can use your data in mathematical expressions and how to list the results to your terminal. Now, we're ready to expand your horizons.

First, you'll learn how to store large amounts of **DATA** internally and **READ** it into variables for computation. Then, we'll move outside your computer and introduce you to devices and files. In fact, we'll show you how devices and files look the same to BASIC09 because of the unique Input/Output structure of the advanced OS-9 operating environment it lives in.

You'll learn the meaning of these BASIC09 statements:

CLOSE	**CREATE**	**DATA**	**DELETE**
GET	**INPUT**	**OPEN**	**PRINT**
PUT	**READ**	**RESTORE**	**SEEK**
SIZE	**WRITE**		

READING AND STORING INTERNAL DATA

Getting your data from the terminal's keyboard is ok when you're dealing with a small amount of information. But, how would you like to enter the name of every player on a baseball team — every time you compute batting averages?

This is a good place for you to use BASIC09's internal **DATA** storage. In fact, **DATA** statements containing the names of a baseball team might look like this:

```
    DATA "Powers","Jones","Puckett"
    DATA "Cobb","Robinson","Bench"
100 DATA "Garvey","Brock","Jackson"
```

Before you can use this **DATA,** you must **READ** it into an array. Here's a procedure that both **READs** and **PRINTs** it.

```
PROCEDURE readplayernames
DIM player(9):STRING[15]
DIM count:INTEGER

    DATA "Powers","Jones","Puckett"
    DATA "Cobb","Robinson","Bench"
100 DATA "Garvey","Brock","Jackson"

FOR count := 1 to 9
  READ player(count)
  PRINT player(count)
NEXT count

END
```

You should understand several things about DATA and **READ** statements before you use them. First, BASIC09 evaluates each expression as it **READs** it from the **DATA** list. The resulting value may be of any **TYPE.** However, the variable that stores the **DATA** must be of the same **TYPE.**

Secondly, you must supply a list of variables to store information read from the **DATA** list. In our example, **DATA** is **READ** from the list and stored in an array. Powers, the first name in the list, is stored in the first element of the array, player(1); Jones, the second name, is stored in player(2), etc.

After all expressions in a **DATA** statement have been **READ,** BASIC09 reads from the next **DATA** statement. When all **DATA** statements in a procedure have been used, the next **READ** statement goes back to the first **DATA** statement and starts through the list again.

But what happens when you need to backtrack and **READ** a specific part of the DATA list again? No problem, just use the **RESTORE** statement. It takes two forms:

```
RESTORE
RESTORE 100
```

Stop for a moment and see if you can figure out what would happen if you applied these statements to the **DATA** in the procedure readplayernames.

That's right, a **READ** statement following the first line would read the name "Powers". When a **RESTORE** statement has no line number, the next value is read from the first **DATA** statement in the procedure.

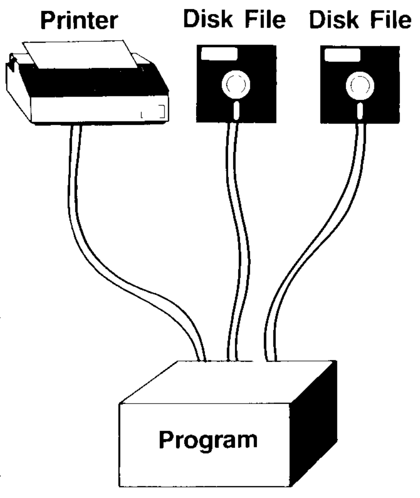
Which name do you think would be read by a **READ** statement following the second **RESTORE** statement?

If you said, "Garvey", you're right. A **RESTORE** statement with a line number causes the next value read to come from a **DATA** statement with that line number.

ALL ABOUT PATHS AND FILES

A file is a sequence of data that has been stored for future use. With BASIC09, data can be of any **TYPE** — the file system doesn't care.

Some files store ASCII text, some hold binary numbers, and others store special codes designed by the programmer. Again, the file system doesn't care.



PATHS

A terminal, modem, and disk file all look the same to BASIC09. They all send and receive data, one character at a time — through a data path.

Paths are data channels used by BASIC09 to tell the OS-9 operating system which device driver to use when sending or receiving your information. A device driver is a piece of software that lets OS-9 talk to a piece of hardware. The name of a path is often the name of a device descriptor that tells OS-9 which device driver to use.

For example, most OS-9 systems use a device descriptor named /p to send data to a printer. This descriptor tells OS-9 to use a device driver named **PIA** and gives it the address of the hardware. Another device descriptor named /t1 talks to a second terminal. It points to a device driver named **ACIA.**

Even disk drives have device descriptors. **/D0** is a common name for the first drive in a system.

Disk drives read data from a magnetic material on a floppy disk. The name and location of each file on the disk is stored in a directory. When you want to read data from a file on a disk, you give BASIC09 a pathlist which contains the device name, a directory name, and a file name. For example:

```
SHELL "list /d1/BOOK/Chapter_10"
SHELL "list Chapter_10"
```

In the first example we use the BASIC09 SHELL statement to give OS-9 everything it needs to know about the file we want to list. We name a drive, /d1; a directory on that device named **BOOK;** and a file in that directory named Chapter_10. In the second example, we assume that the file Chapter_10 is stored in the current data directory.

If filenames, pathlists and data directories are still a bit hazy, now would be a good time for you to review our introduction to OS-9 in Chapter Three. You'll soon be back on the right track.

With the OS-9 operating system, your imagination is the only limit. By redirecting the standard input and output paths, you can test a terminal input routine with a disk file. Or, you can look for errors in a disk file procedure by typing in characters from the keyboard.

But for now, let's keep it simple and use the standard input and output paths to **INPUT** data from your keyboard and **PRINT** it on your terminal.

INPUTTING DATA FROM YOUR TERMINAL

Sometimes, you need to pass some information to your computer while it is running a program. How do you do it?

The first thing you must do is let the operator know that the program needs some information by using a prompt.

After your program prints a prompt, it must stop and wait for an answer. You tell it to wait by using an **INPUT** statement. In fact, BASIC09 lets you kill two birds with one stone. You can issue the prompt and get the information with one statement.

For example, suppose that you are writing a program for a medical clinic. Each time a new patient visits, the doctor wants the patient's name added to a file. Here's a statement that will ask the operator for the patient's name and **INPUT** it into your program:

INPUT "What is the patient's name? ", name$

At other times you may be dealing with operators who know how to use a computer. Often — especially when they must run the same program many times a day — they don't want to be slowed down by prompts. To keep them happy, you can use the simplest form of the **INPUT** statement. Be aware that the practice of not using prompts is a bad habit and if they forget what the program wants, they may wind up a victim of this shortcut.

INPUT number, points

When you use the **INPUT** statement above, your computer will not print an English language prompt. Instead, it will print a question mark, "?". When you see the question mark you can be sure that your computer wants some information. But, it will be up to you to remember what it wants.

You may want to have your program get its information from another terminal. To do this, you **OPEN** a path to that terminal and use the new path number in your **INPUT** statement.

```
OPEN #newterminal, "/t1":UPDATE
INPUT #newterminal, number, points
```

We'll describe the **OPEN** statement in detail later in this chapter. For now, we have one more point to consider. Did you ever wonder what happens to data after you **INPUT** it?

Data read into BASIC09 with the **INPUT** statement is handled like data **READ** from an internal **DATA** statement. The data you type on the keyboard is assigned to variable names in your input list in the order the names appear.

When entering data, you must type all requested variables — separated by commas — and then type <**RETURN**>. If you make a mistake, this message will appear.

```
**INPUT ERROR—RETYPE**
```

If you see this message, you must re-type the entire line. Don't get too upset. Here's a tip that will make life easier.

BASIC09 uses OS-9's line **INPUT** statement which means that if you make a mistake, you may edit the line — up until the time that you hit the carriage return.

You may backspace to correct your error, delete the entire line, repeat an entire line, or send an end-of-file character. Now would be a good time to review the line editing features we introduced in Chapter Two.

PRINTING DATA ON YOUR TERMINAL OR PRINTER

Since **PRINT** is one of the best known verbs in the BASIC language, we won't dally here too long.

The **PRINT** statement sends the value of each item in an output list to the standard output device. Nine times out of 10, the standard output device will be your terminal. You may however, send information to an alternate device by inserting a path number in your **PRINT** statement.

Here are the rules. Items in your output list must be separated by a comma or a semicolon.

If you use a comma, each item is aligned in a tab zone, which means an item will be printed every 16 columns across the screen. The first item will start at the left-hand side of the screen, the second begins at column 16, the third at column 32, the fourth at column 48, and the fifth at column 64. Try this!

```
FOR number := 1 to 12
  PRINT number,
NEXT number
```

Your screen should look like this:

```
1.              2.              3.              4.
5.              6.              7.              8.
9.              10.             11.             12.
```

Now, change the comma to a semicolon and see what happens. You should see something like this:

```
1.2.3.4.5.6.7.8.9.10.11.12.
```

When you use a semicolon in a **PRINT** statement, all of your data will run together — there will be absolutely no spacing between items. Also, if you put a semicolon after the last item in a list, the carriage return will be inhibited and the cursor will stay on the same line. You'll use this trick often.

If you are only **PRINT**ing one item and you do not use a comma or a semicolon, BASIC09 sends a carriage return after it prints the item. The cursor will be moved to the next line on the screen. Omitting the comma or semicolon after the last item in a list has the same affect.

Here are some more examples:

```
PRINT location, time, temp
PRINT name, address, zip
PRINT name; " "; address; "; zip
```

The first example **PRINTs** three numeric variables. The number of the location computed should appear at the left edge of your screen, the time in column 16, and the temperature in column 32.

The second example prints two variables of type **STRING** and one of type **REAL.** You expect them to be printed at the left edge of the screen, at column 16 and column 32 respectively. But, what happens if the strings are longer than 16 characters?

If "name" is longer than 16 characters, it extends into the second zone, and "address" is printed at column 32 instead of column 16. The same is true with "address" — if it is longer than 16 characters, "zip" will be print in the next tab zone.

The final example prints the same information using semicolons. We used two quote symbols, ", to print a space between the items so "name" and "address" would have a space between them.

A character or characters enclosed in quotation marks is treated as a string constant. String constants are also known as a literal strings. Essentially, what you see is what you **PRINT.** Try it!

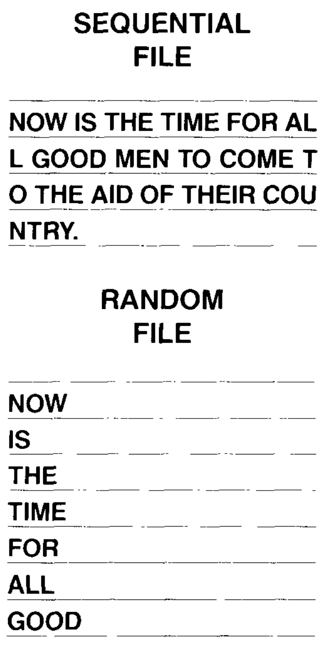
PRINT "This is sure easy!"

Professional presentations are the result of paying close attention to **PRINT** formatting. BASIC09 has several powerful statements — including **TAB, POS,** and **PRINT USING** — to give you the additional control needed. We'll cover them in detail in the Chapter 11.

TWO TYPES OF FILES — RANDOM AND SEQUENTIAL

BASIC09 uses two types of files: sequential and random-access.

Sequential files hold records. Records, hold ASCII characters. There can be any number of characters in a record and any number of records in a file — as long as they will fit on the disk.



It is easy to picture a record if you think of it as a line of text. The character that most often marks the end of a record is a carriage return.

To put a record in a sequential file, you use a **WRITE** statement. It sends each character in your record to the file and then sends a carriage return. To get information out of a sequential file, you **READ** it. The **READ** statement reads characters from a file until it finds a carriage return.

Random access files work differently. They hold a mirror image of BASIC09 data. Data written to a disk file looks just like the same data in memory. There are no carriage returns to indicate the end of a record.

Since information doesn't have to be converted to ASCII before it is written to a random access file, operation is very fast. The same is true when the data is read from a disk.

To store data in a random access file, you use BASIC09's **PUT** statement. To retrieve it, you use the **GET** statement.

SEEK and **SIZE** statements help you **PUT** your record in the right place in a random access file. They also help you **GET** the proper record from a file.

SEEK points to a specific character in a file. For example, **"SEEK 0"** points to the first character in a file, **"SEEK 100"** to the 100th, etc.

SIZE tells you the size of a data element and gives you a way to calculate the position of any record in a file.

Before you can **WRITE, READ, PUT, GET, SEEK,** or **SIZE** a disk **FILE** — you must **CREATE IT.**

LET'S CREATE A FILE

To create a new file on your disk, you use the **CREATE** statement.

CREATE #draft, "/d1/worktext/draft":WRITE

This statement creates a file named "draft" in a directory named worktext on the disk you have loaded in drive, /d1. The path number assigned to the file is stored in the variable "draft". Later, you will use the name "draft" when writing to the file. The variable holding the path number must always be of type **BYTE** or **INTEGER.**

The **WRITE** following the name of the file tells BASIC09 what you are going to do with the file. If you use the statement above, you will not be able to read from the path, "draft". You may only **WRITE** to it.

There are three access modes you may use when you **CREATE** BASIC09 files.

WRITE **UPDATE** **EXEC**

You can create a file for **WRITE** only as you did above.

You can create a file in the **UPDATE** mode when you need to both **READ** and **WRITE.**

And, if you need to store machine langauge code that will be executed, you can use the **EXEC** mode. This mode is very rarely used except by expert Basic09 gurus.

Both sequential and random access files are **CREATED** the same way. They look exactly the same on the disk. When a file is created it has a length of zero. As you **WRITE** to the file, it expands automatically.

Although there is probably no good reason for you to do it, you may also **CREATE** a file on a device — /t1 for example — without causing an error. BASIC09 will treat the **CREATE** statement like an **OPEN** statement.

LET'S OPEN A FILE

If a file already exists you can skip the **CREATE** statement. You must, however, **OPEN** a path to a file before you can use it. Here's one way to **OPEN** a path and send a message to your printer.

```
DIM print_path:BYTE
DIM name:STRING[2]

name := "/p"

OPEN #print_path,name:WRITE
PRINT #print_path, "The Quick Brown Fox Jumped ... "
CLOSE #print_path
```

You should know several things about the **OPEN** statement. First, the variable holding the path number — print_path in our example — must be of type **BYTE** or **INTEGER.** And, you have more access modes available to you when you **OPEN** a file than when you **CREATE** one.

READ WRITE UPDATE EXEC DIR

READ lets you read from the file, **WRITE** lets you write to it, and **UPDATE** lets you either **READ** or **WRITE.** In fact, if you do not select an access mode when you **OPEN** a file, BASIC09 uses the **UPDATE** mode automatically.

You can combine these access codes using a + as shown below:

```
OPEN #dirpath,dirname:READ + DIR
```

Note that **READ + WRITE** is the same as **UPDATE.** You can similarly combine legal access codes in **CREATE** statements.

If you **OPEN** a file using the **EXEC** mode, BASIC09 stores or looks for your file in the current execution directory. BASIC09 uses the current data directory for all other modes.

The **DIR** mode **OPENs** a directory for **READ.** BASIC09 will not let you **OPEN** a directory file in the **WRITE** or **UPDATE** modes, and, if you try it, BASIC09 will print an error message. If BASIC09 did let you do this, the damage you would do to your disks just testing your programs would make you cry!

LET'S WRITE TO OUR FILE

The **WRITE** statement should look very familiar. In fact, it's almost like the **PRINT** statement. Here's the difference.

When using the **WRITE** statement, you must use a path number. With the **PRINT** statement, the path number was optional. You must always use commas to separate items in your data list when using **WRITE.**

```
WRITE #printer_path, lastname$; ", "; firstname$
WRITE #1, LEFT$(firstname$,1); ". "; lastname$
```

The first statement **WRITEs** a lastname, a comma, a space, and a firstname to your printer. The name written is the one you stored earlier in the variables named lastname$ and firstname$.

The second example sends the first letter of the name held by firstname$, a period, a space, and the name held by lastname$ to the standard output device — usually your terminal.

Here are some additional things you should know about **WRITE.** This statement always sends data to a file in ASCII character format. The path number is usually determined when the file is **OPEN**ed or **CREATE**d and is stored in the **BYTE** or **INTEGER** variable you named then.

Again, when you type the data list in a **WRITE** statement you must separate the items with commas. Items are written one after the other. When you are writing more than one item in a record, BASIC09 places a null — a character with a value of 0 — between items.

Expressions in your list may evaluate to any data **TYPE.** However, BASIC09 converts them to ASCII before writing them to your file.

Records created with **WRITE** statements vary in length. If you **WRITE** a single character to a record, that record will only be one byte long. If you **WRITE** a **STRING** 32 characters long to a record, the record will be 32 bytes long, etc.

There's one more thing you need to know. OS-9 keeps a file pointer in memory that tells it where to find the next character in a file. This pointer is updated automatically every time data is written. When you use a **WRITE** statement, BASIC09 always starts writing at the location of the pointer.

Once you have written a record to a file, you are home free. You can **READ** it back anytime you need it. Let's move on.

READING OUR RECORDS

Sometime after you have written data to a disk file, you're going to need to use it. To get it back in the computer, you use the **READ** statement? Let's jump right into some examples.

```
READ #myfile,name$,address$,city$,state$,zip

PRINT #1, "Height, Weight?  "
READ #0, height, weight
```

The first statement **READs** five items from a record pointed to by a path named "myfile". It **READ**s one item at a time and stores it in the appropriate variable.

Always remember, once you have written a number of data items to a file in a certain order, you must **READ** them back in the same order.

The second example shows how to **READ** data from your terminal. Instead of using the name of a variable for the path number, we typed #0. By using #0, BASIC09 knows to **READ** the data from the standard input device — usually the keyboard on your terminal.

As you would expect, the **READ** statement is the exact opposite of the **WRITE** statement. It's operation is identical.

For example, a **READ** also begins at the current location of the file pointer. The file pointer is updated as the data is read. Sure sounds familiar!

Again, the path number is usually held in a variable initialized earlier by an **OPEN** or **CREATE** statement. **READ** works with files **OPEN**ed for **READ** or **UPDATE.**

Items from a record are stored in the variables listed after the path number in the **READ** statement. They are stored in the order read.

READ expects **STRING**s in a record to be separated by a null character. However, numeric data may also be separated with a comma or space. **READ** expects a carriage return at the end of each record.

READ and **WRITE** were designed to work together. Everything **READ** expects, **WRITE** does. The moral of the story — when you plan to get data from a file with a **READ** statement, make sure you put it there with a **WRITE** statement.

DON'T FORGET TO CLOSE YOUR FILE

What's that old saying? The job isn't done until the paperwork is complete.

There's a parallel in programming. You aren't through with a file until you **CLOSE** it.

Granted, BASIC09 is pretty forgiving. Most of the time it **CLOSE**s your files for you if you forget, leaving files **OPEN** is a very bad habit. It ties up memory that can be used elsewhere, and if you forget to **CLOSE** the path to a non-sharable device — a printer or modem for example — you are keeping another person from using it.

Besides, the syntax is simple. Very little effort is required.

```
CLOSE #masterfile,#myfile,#yourfile
```

The line above hardly requires an explanation. It closes the three files **OPEN**ed on the path numbers named.

It's a good idea not to **CLOSE** the standard input, output and error paths (#0, #1, and #2). However there's nothing to stop you, when you have a good reason.

For example, if you want to redirect the standard output path from your terminal to your modem, you could use these two lines. The following example is somewhat dangerous because if an error occurs, BASIC09 would try and communicate with you via the modem!

```
CLOSE #1
OPEN #path, "/modem"
```

There's only one catch to the **CLOSE** operation — the path you are closing must be **OPEN.** When you stop to think about it, that makes sense. Let's show you how to get rid of a file, then we'll move on to demonstrate the power of BASIC09's random access files.

TO GET RID OF A FILE — DELETE IT

When you're sure you're ready to bid farewell to a file forever, you can **DELETE** it.

DELETE is a simple command, but one that must be used with caution. Once you **DELETE** a file, its name is removed from the directory, its storage deallocated, and its data lost forever.

You may use either a literal string or a string variable in your **DELETE** statement, and you may use a computer pathlist or the filename alone.

But remember, if you do not give BASIC09 the complete pathlist, it will assume that your file is in the current working directory.

```
DELETE "/d1/worktext/Chapter_11"

filename$ := "myfile"
DELETE filename$
DELETE "/d1/" + filename$
```

The first statement deletes a file named Chapter_11, located in a directory named worktext on a disk installed in drive /d1.

The second uses a string variable to **DELETE** a file named myfile. BASIC09 expects to find it in the current data directory.

The last example shows how you can add a string variable to a literal string to create a pathlist for the **DELETE** statement.

Here's one more thing you must remember. If you do not own a file, you cannot **DELETE** it. In fact, you can't even **WRITE** to it.

If you're confused about file ownership, now would be a good time to review the introduction to OS-9 operating system in Chapter Three. Otherwise, we'll move on to random access files.

WHEN YOU PUT DATA IN A FILE, YOU CAN GET IT FASTER

In this section you'll be introduced to one of the most important advantages of BASIC09 — the ability to **GET** and **PUT** any amount of any **TYPE** of data in a single statement. In fact, we'll present a few program lines that show you how to **GET** an entire data structure in two lines. That's something you can't do with too many BASICs today.

If you've forgotten about the user defined data structures introduced in Chapter Seven, take the time now to review them. After you're finished, you'll appreciate the power we're unleashing even more.

Once you learn to combine BASIC09's complex data structures with its random access file statements, you'll be amazed at the results. Your procedures will be self-documenting. You'll be able to tell what a procedure is doing — just by reading the code. The descriptive names you give to your data elements make it easy.

Since BASIC09 **PUT**s data in a file in exactly the same form that it stores it internally, the **PUT** statement operates fast. It doesn't need to stop and convert your data to and from ASCII like the sequential **READ** and **WRITE** statements. Your programs will be much faster and simpler than your neighbor's.

And last but not least, you can **GET** and **PUT** data at any place you specify in a file, not just sequentially as with **READ** and **WRITE.** Being able to **GET** and **PUT** data in any order (called "random access") can vastly simplify and speed up data retrieval programs.

First, the rules. BASIC09's **GET** statement reads fixed-size records from a file and stores them in a variable, array, or complex data structure.

The **PUT** statement does just the opposite. It gets fixed size data from a variable, array, or complex data structure and writes it to a file.

We'll review BASIC09's data types here and show you what **GET** and **PUT** will do with each. Let's start with data of type **BYTE** — one character.

When you **GET** a **BYTE** from a file, BASIC09 reads the character pointed to by OS-9's file pointer. **PUT** does just the opposite — it writes one character to your file.

What do you suppose happens when we **GET** or **PUT** an **INTEGER.** Did I hear you say it reads or writes two characters? You've already got the idea!

When you **GET** or **PUT** a **REAL,** you are reading or writing five bytes at a time. Likewise, when you **GET** or **PUT** a **STRING,** you read or write the number of bytes you specified when you **DIM**ensioned the **STRING.**

Just like sequential files, random access files must be **CREATE**d or **OPEN**ed before you can **PUT** data in them or **GET** data from them. Also if you plan to **GET** data, you must **OPEN** the file in the **READ** or **UPDATE** mode. Conversely, if you need to **PUT** data in a file, you must **OPEN** it for **WRITE** or **UPDATE.**

You can **PUT** data in a file from only one variable, array, or data structure at a time. Thus, it follows that you can only **GET** the data for one variable, array, data structure with each **GET** statement.

The big difference between **GET/PUT** and **READ/WRITE** lies in the form in which they transfer their data to or from a file. Remember, the **WRITE** statement converts your data — no matter what **TYPE** —to ASCII before it sends it to a file. Because of this difference, the **READ** statement must convert the data back from ASCII to the proper **TYPE** before storing it.

Those are the basic rules for **GET** and **PUT.** Now, let's unveil their power. Since you can **PUT** an entire data structure into a file with one statement, you save work and improve the readability of your programs at the same time.

Pretend for a moment that you own an electronics parts company and you need to set up a file to handle your inventory. You need to store the name of an item, its cost, its selling price, and the number you have on hand.

Inventory Record

Item Name	Cost Each	Selling Price	Number on Hand
Widget	13¢	29¢	5,280

MAKEFILE

```
PROCEDURE makefile

        TYPE inventory_item = name:STRING[25];list,cost:REAL; qty:INTEGER

        DIM inventory_array(100):inventory_item
        DIM work_record:inventory_item
        DIM path,counter:BYTE

        CREATE #path, "inventory"

        work_record.name := ""
        work_record.list := 0.
        work_record.cost := 0.
        work_record.qty := 0
        FOR counter := 1 to 100
          PUT #path, work_record
        NEXT counter
        END
```

Let's step through the example procedure a line at a time.

We start with one statement that defines a new data **TYPE** named "inventory_item". This new data **TYPE** is made up of a **STRING** containing 25 characters, a **REAL** variable holding the list price of each item, another **REAL** variable holding the cost of the item, and an **INTEGER** variable holding the quantity of the item in stock.

Next, we reserve space in memory with the **DIM** statement. Our procedure reserves enough memory for two data structures — an array large enough to hold 100 elements of **TYPE** "inventory_item" and a "work_record" large enough to hold one "inventory_item".

After we reserve memory for our data, we **CREATE** a path to a file named, "inventory".

Then, we initialize the file to a known value. The next several lines do the initialization. First, a null **STRING** is stored in the name field of the "work_record". Then, the list, cost, and quantity fields of this record are set to a value of zero.

At this point, only the data fields in memory are initialized. We must still **PUT** our initial data into each record in the file. A **FOR** ... **NEXT** loop does the job.

Now that we have initialized our file, how do we **GET** our data back to verify the fact that the file is initialized?

There are several ways to do it. Your first reaction would be to use a **FOR/NEXT** loop to **GET** each record from the file and store it in an array. That would work fine. But there's a much easier way to do it with BASIC09.

We can simply read in the entire structure, inventory__array, with one statement. Try it!

```
SEEK #path,0
GET #path,inventory__array
```

SEEK positions OS-9's file pointer. In the first line above, the pointer is set to the beginning of the file. The effect is similar to the **RESTORE** statement used with internal **DATA.**

The second line reads in the entire array with one statement. This method is much easier than using a **FOR** ... **NEXT** loop and much faster.

Now you know how to position the file pointer at the beginning of your file. But what do you do if you need to **GET** the tenth record. Enter another new word, **SIZE.**

SIZE is a BASIC09 function that returns the number of bytes of memory reserved for a variable, array, or complex data structure. For example, if you had **DIM**ensioned a variable named "price" as type **REAL,** the call **SIZE**(price) would return a value of "5" since it takes five bytes of memory to store a **REAL** number.

In the case of a **REAL** number you would have known the size anyway. But, how about complex data — such as inventory__item or inventory__array? Let's look at the **TYPE** statement in the procedure, makefile, closely.

The "name" field requires 25 bytes. Then, add 10 bytes for the two **REAL** numbers, list and cost, and two more for the **INTEGER,** qty, and you have a total of 37. The size of an inventory__item, or one record is 37. The size of the entire array of 100 inventory__items would be 3700.

This time it wasn't too hard to calculate the size of our record. However, the more complex the data type, the longer it will take you to calculate the size. The chance for error also increases. It's much easier to type "**SIZE**(inventory__item)".

Back to our problem. To find the location of the tenth record or inventory__item, we must calculate how many bytes we have used to store the first nine records. If we were calculating the position by hand we would multiply 37 times nine to give us 333. We would then use the statement, **"SEEK #path, 333".**

The process described above could become a real drag if you had to do it every time you needed a record. The easy way to solve the problem is to use this statement:

```
SEEK #path, 9 * SIZE(inventory_item)
```

This statement calculates the position of the tenth record in the file for you automatically. Here's a more general statement that can be used to find any item in the file:

```
SEEK #path, (item_number—1) * SIZE(inventory_item)
```

Once you have initialized your file, you will want to **PUT** information in it. To do the **PUT,** you need only prompt for each item in the record, use the **SEEK** statement to position the file pointer and then **PUT** the record in the file. A sequence of statements like this does the job.

```
INPUT "Item number? ",item_number
INPUT "Item name? ",work_record.name
INPUT "List price? ",work_record.list
INPUT "Cost price? ",work_record.cost
INPUT "Quantity? ",work_record.qty

SEEK #path, (item_number—1) * SIZE(work_record)
PUT #path,work_record
```

SUMMARY

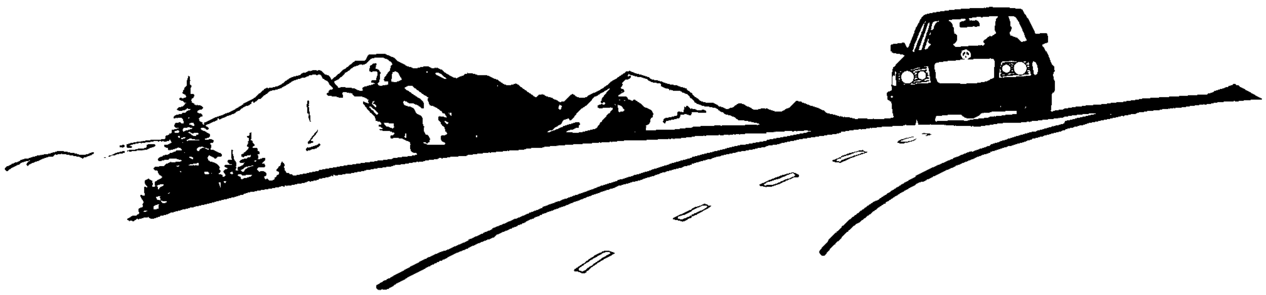
You're almost home free. When you add the BASIC09 statements you've learned in this chapter to those you already know, you should be able to do just about anything.

In this chapter you've learned how to let your programs communicate with the outside world. You should know how to **READ** information from internal **DATA** statements or your keyboard and how to **PRINT** data on your screen and printer.

We've introduced you to sequential files and shown how to **CREATE,** and **OPEN** them. We've shown how to **WRITE** data to files and **READ** from them. And, we've shown you how to store complex data structures with single **GET** and **PUT** statements and how to find individual records in them with the **SIZE** and **SEEK** statements.

CHAPTER ELEVEN

who says form follows function



Hold on to your hats. We've got a fast track and your Mercedes should make this short chapter in no time!

The debate concerning the rightful place of form and function in the world is as old as man. We won't try to sway you one way or the other. But if you're the type that thinks function follows form, you'll love this chapter.

The commands you learn here will let you make your reports look good on the terminal's screen or on the printed page. You'll be able to format data till your hearts content. We'll introduce you to:

TAB **POS** **PRINT USING**

YOU CAN TAB TO ANY POSITION

BASIC09's **TAB** statement gives you a way to **PRINT** your data in columns. It causes the cursor head to move to the column you name. For example,

```
PRINT "Column #25"; TAB(25); "*"
```

This line will cause BASIC09 to **PRINT** Column #25 along your terminal's left hand margin and a star, "*", in column 25. Give it a try.

If BASIC09 attempts to execute a **TAB** statement but finds that the cursor or print head is already past the column position requested, it simply ignores the **TAB.**

Here's a bit of trivia that will come in handy if your printer can print long lines. BASIC09 output columns are numbered between one and 255.

POS WILL TELL YOU WHERE YOU ARE

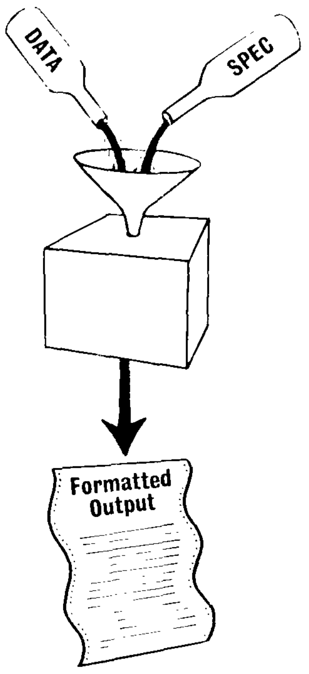
Another handy formatting statement is **POS.** It returns the present cursor position. For example:

IF POS > 31 THEN PRINT

You can use this line when you need to **PRINT** to a screen that only displays 32 columns — like the Radio Shack Color Computer. It causes BASIC09 to issue a carriage return and linefeed when the cursor moves past the 31st column. There are many other uses for the **POS** statement. Turn your imagination loose!

PRINT USING GIVES YOU COMPLETE CONTROL

If you have no use for form and are quite content with a screen filled with numbers printed in random locations — feel free to skip the rest of this chapter. You'll find no need for pretty printing. If however you like things neatly organized with all your rows and columns lined up like soldiers, standby. Your dream is about to come true.



PRINT USING makes it easy to generate slick reports that are sure to please your boss. But before we give you the details, let's show you the difference between **PRINT** and **PRINT USING.** Enter and **RUN** the procedure **DEMO** in Chapter One. Notice how slick the multiplication table is lined up.

Now, enter the edit mode and change the line containing the **PRINT USING** statement so that it uses a normal **PRINT** statement. Then, **RUN** the program again. Wow, what a difference! Convinced? Let's move ahead.

PRINT USING normally sends its output to the standard output device — your terminal. When you want it to output to a printer or disk file, you need only add an optional path number to your statement.

PRINT #es USING "S80^", "Hello Esther!"

You tell your computer how you want it to format your information by including a format specification between the words **PRINT USING** and the data. For example, the "S80^ " above tells BASIC09 to center the message in a line 80 characters long.

The format specification tells BASIC09 the **TYPE** of number you will be printing, the width of the print field, and how you want the data arranged. You can line up the left margin, the right margin, or center a column of data.

As BASIC09 sends your data to the terminal it matches up each format specification with an item in the output list. If it runs out of format specifications before it runs out of data, it goes back to the first format specification and starts through the list of again.

Here's another detail you must watch closely. The **TYPE** of data called for in the format specification must agree with the **TYPE** of data in the output list.

A format string may contain one item or several. Also, when you need to specify more than one format in a line, you must separate each format specification with a comma.

You can pass two kinds of format specification to BASIC09. The first controls the format of an item from your output list. The second sends its own data. Examples of the second type are the codes that send tabs or a string of characters to your terminal.

Format specifications are made up of a letter, a number, and a special character that tells BASIC09 how to justify the data.

The letter tells BASIC09 what type of data it can expect to find in the output list. One of six letters is used.

R	=	**REAL, INTEGER or BYTE data (decimal format)**
E	=	**REAL, INTEGER or BYTE data (exponential format)**
I	=	**INTEGER or BYTE data**
H	=	**HEXADECIMAL format of ANY data type**
S	=	**STRING format**
B	=	**BOOLEAN format (prints TRUE or FALSE)**

The number following the letter is a constant that tells BASIC09 the field width. Here's how it works.

When you tell BASIC09 that you want a field width of 10, you are telling it to allow 10 column positions for this field — no matter how many positions it takes to hold the data.

When you pick a field width you must consider the maximum size of your data and allow a few extra positions. For example, a **REAL** number printed in decimal format needs extra spaces for the decimal point and sign character.

When you print in the exponential format you must leave space for the mantissa sign, a decimal point and exponent characters.

The justification character can be a less than sign, "<, a more than sign, ">", or an up arrow, "↑".

A "<" causes the data to be justified to the left with a leading sign and trailing spaces.

A "<" causes the data to be justified to the right with leading spaces and sign.

The up arrow, "∧", causes different results with each format.

Used with real numbers, the "∧" gives you a financial format. The field is right justified with leading spaces and a trailing sign character.

Used in the **INTEGER** format, the "∧" right justifies the data with leading sign and zeros.

And finally, used in the **HEXADECIMAL, STRING,** or **BOOLEAN** formats, the "↑" simply centers the data in the field. Here are some examples.

```
                FORMAT                    PRINT POSITIONS
             SPECIFICATION                  1234567890

PRINT USING "R8.2<", 5678.123               5678.12
PRINT USING "R8.2>", 12.3                      12.30
PRINT USING "R8.2>", -555.9                  -555.90
PRINT USING "R10.2↑", -6722.4599              6722.46-
PRINT USING "H2<", 100                      C4
PRINT USING "H4<", 100                      00C4
PRINT USING "S8<", Hello                    Hello
PRINT USING "S8>", Hello                        Hello
PRINT USING "S8↑", Hi                          Hi
```

These examples give you a good idea of how the **PRINT USING** statement works. Use the numbers above the last column to compare the effect of different format specifications.

Let's look now at the second form of format specification — the type that actually sends its own data to your screen. BASIC09 gives you three control specifications to control horizontal formatting. They may be used anywhere in a **PRINT USING** statement.

T6	**TABS to column 6**
X6	**Outputs six SPACES**
"string"	**PRINTs the word, "string"**

Here's an example:

```
PRINT USING "addr",X2,H4,X2,"data",X2,H2",1000,100
```

The output from the line above looks like this:

```
addr 03E8 data C4
```

BASIC09 also lets you repeat a sequence of specifications several times by typing a repeat count followed immediately by the specifications you want to repeat. The specifications are placed in parentheses.

Both of these lines print the same way.

"2(X2,R8.2>)" "X2,R8.2>,X2,R8.2> "

PRINT USING statements really make your data look professional. To see the result, **RUN** these sample programs.

PRIMES

```
PROCEDURE primes
     0000      (* A program to compute and tabulate)
     0024      (* the first N prime numbers. *)
     0044
     0045      (* CONSTants *)
     0054      DIM nprimes,sqrtnp:INTEGER
     005F      nprimes=400
     0067      sqrtnp=20
     006E
     006F      (* VARiables *)
     007E      DIM prime:BOOLEAN
     0085      DIM i,j,k,lim,x,square:INTEGER
     00A0      DIM p(400):INTEGER
     00AC      DIM v(20):INTEGER
     00B8
     00B9      p(1):=2
     00C3      x:=1
     00CA      lim:=1
     00D1      square:=4
     00D8
     00D9      FOR i:=2 TO nprimes
     00EA
     00EB        REPEAT
     00ED          x:=x+2
     00F8
     00F9          IF square<=x THEN
     0106            v(lim):=square
     0112            lim:=lim+1
     011D            square:=p(lim)*p(lim)
     012F          ENDIF
     0131
     0132          k:=2
     0139          prime:=TRUE
     013F
     0140          WHILE prime AND k<lim DO
     0151
     0152            IF v(k)<x THEN
     0162              v(k):=v(k)+p(k)
     0178            ELSE
     017C              prime:=x<>v(k)
     018B              k:=k+1
     0196            ENDIF
     0198
     0199          ENDWHILE
```

```
019D
019E          UNTIL prime
01A6
01A7          p(i): = x
01B3
01B4          IF MOD(i,10) = 0 THEN
01C3            PRINT
01C5
01C6            FOR j: = i-9 TO i
01DB              PRINT USING "I6> ",p(j);
01EB            NEXT j
01F6
01F7          ENDIF
01F9
01FA        NEXT i
0205
0206        PRINT \ PRINT
020A
020B        END
020D
020E
```

The procedure primes prints the first several hundred prime numbers in 10 neatly spaced fields exactly six columns wide. The table comes in handy every once in a while too. It may even win the most insignificant trivia contest at a party.

Here's one that should really interest you. Want to get rich?

FUTURVAL

```
PROCEDURE futurval
0000      DIM value,principal,interest:REAL
000F      DIM timescompounded,years:INTEGER
001A
001B      PRINT "Let's figure the future value of an investment! "
004F
0050      REM First we need to ask a few questions.
0078
0079      INPUT "What is your initial investment? ",principal
00A2      INPUT "OK, What is the nominal interest rate? ",interest
00D1      INPUT "How many times will interest be compounded? ",timescompounded
0105      INPUT "How many years? ",years
011D
011E      interest = interest/timescompounded/100
012F      value = principal*(1 + interest)/(timescompounded/years)
0148
0149      PRINT "Your future value is $"; INT(value*100 + .5)/100
0177      PRINT
0179
017A      END
```

To run this program just type **RUN** and answer the prompts. Happy Daydreaming!

SUMMARY

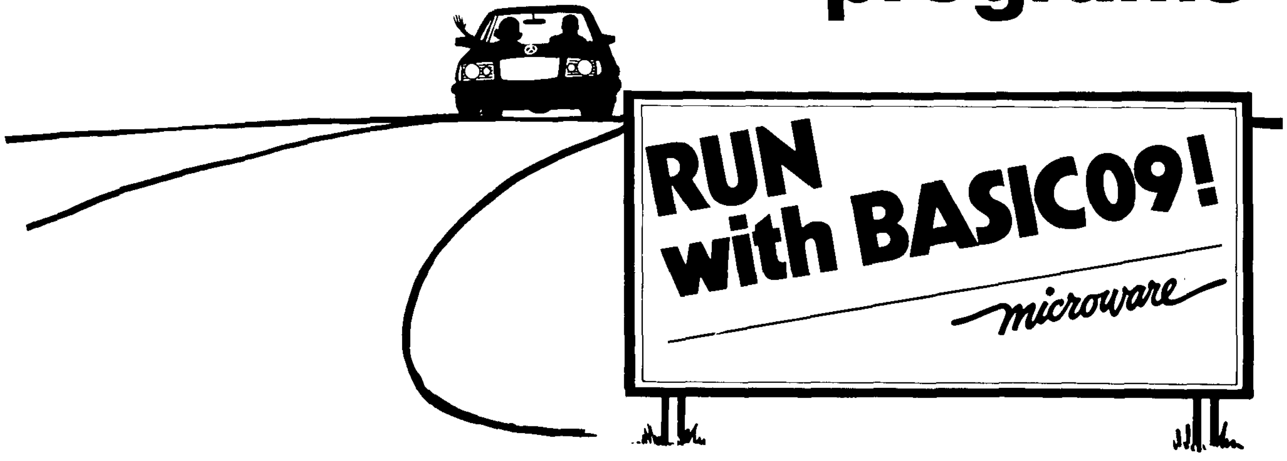
In this chapter we introduced you to **TAB, POS,** and **PRINT USING.** You should be able to print a report that really dazzles the boss.

We promised you a short chapter. I hope you weren't disappointed. Don't hang up the keys to your Mercedes yet — we only have two chapters to go. You're well on your way to mastering BASIC09.

In Chapter 12 we'll show you how your BASIC09 procedures can **RUN** other procedures automatically. See you there!

CHAPTER TWELVE

letting BASIC09 run its own programs



In Chapter Five we taught you how to drive your Mercedes with a stick shift. Working with BASIC09's system mode, you learned a lot of commands and found out how to **RUN** your programs manually. Now, we'll shift to the automatic transmission and show you what BASIC09 can do on its own.

In this chapter you'll learn more about the **RUN** statement and find out how to use it within your programs.

You'll learn how your programs can run many smaller tasks — or procedures — automatically. In fact, we'll be encouraging you to break your problems down into these smaller, more manageable pieces.

You'll be in for a real treat when we show you how to make your procedures run themselves over and over again. It's a technique called recursion that makes it easy for you to solve many tough problems.

And finally, you'll meet parameters. We'll show you how to define them and give you a chance to pass them to a procedure. We'll even show you how to love them interactively as you type them inside a **RUN** command from BASIC09's system mode. Before the chapter ends, you'll be letting all your programs pass them. Don't worry, they're perfectly legal!

MODULARITY

Novels are merely organized collections of well written chapters. Chapters contain a number of well written paragraphs. Paragraphs are collections of carefully constructed sentences. Sentences are just short groups of well chosen words.

Get the idea? Great! So when are you going to write the great American novel? See you at the bank.

The analogy between writing and programming is strong. A good writer uses short words to build short sentences. He organizes those short sentences into powerful paragraphs. In a few days, those paragraphs become a chapter. And in a few months, the chapters become a book.

A writer does not sit down and write a book. Rather, he breaks the subject matter down into logical chapters. That done, he attacks each chapter with vengeance. Another logical division is made and ideas for sub-chapters are researched. When the research is complete an outline of paragraph ideas is put on paper. Finally, sentences are composed, one word at a time.

Today, a similar approach is being used by successful programmers. It's called structured programming.

Your first step as a programmer is to define the problem you are trying to solve in terms of smaller problems. These small problems can then be broken down into yet smaller problems. Eventually you will reach the point where you can translate your problem directly into a statement the computer can understand.

After you have tackled your programming problems in this manner for a while you will begin to see a similarity in the "smaller problems." Then, you'll start to save your solutions and use them again and again. With BASIC09, you'll be saving and running "modules".

Eventually you'll have a library of modules that can be used as building blocks in hundreds of programs. Building blocks are what modularity is all about. BASIC09 calls its modules "procedures" and makes them easy to use.

THE RUN STATEMENT: A PROGRAMMER'S MARATHON

There are more ways to use BASIC09's **RUN** statement than there are marathons. We introduced you to this command briefly in Chapter Five. Now, we need to get you in shape for the big race. Let's review.

If you are operating in BASIC09's system mode, running a program can be as simple as typing **RUN.** You can do this when your program or procedure is self contained and doesn't need any parameters.

Remember, when you type **RUN**, BASIC09 attempts to **RUN** the current procedure. That's the one marked with the star when you look at a **DIR**ectory of your workspace. If you have a number of procedures in your workspace, you must tell BASIC09 which one you want to **RUN**. For example, if you're preparing for a race you might want to "**RUN FASTER**".

PASS THE WORD — USE A PARAMETER

If you want to succeed in business, you must pass the word to your employees. If you want to win the war, you must pass the ammunition. Likewise, if you want your BASIC09 procedures to **RUN** properly, you must pass the parameters.

It's really frustrating when you know how to do something well — how to change a flat tire on your Mercedes for example — but can't make someone else understand. The chauffeur always pleads ignorance and you wind up changing it yourself.

An analogy can be made to programming. If you don't know how to tell the computer what to do, you'll wind up doing it yourself.

For example, if you want to multiply two numbers and print the product, you can approach the problem several ways. You can write a procedure to multiply "2" times "2" and another to multiply "2" times "4", etc. But that would take a lot of time and memory — besides, it would bore you to death.

It would be much easier to write one procedure that can multiply any two numbers and then pass the numbers you want it to use when you need an answer. The numbers you are passing are called parameters.

Never forget to pass your parameters. Remember what happened when you tried to "RUN ROMAN" the first time in Chapter Five? Did you really expect your computer to know which number you wanted to print in Roman numerals — even though you forgot to tell it?

Remember, a parameter is a number, character, or string of characters that is given or passed to another BASIC09 procedure.

It may be a variable name, a string constant or a number. Study these statements.

Using Literals:

```
RUN print_a_word("Hello")
RUN roman(1983)
```

Using Variables:

```
RUN print_a_word(firstword)
RUN roman(year)
```

To make sure we're all on the same track, we'll look at three procedures that show how parameters make life easy.

HARDMULT

```
PROCEDURE hardmult
      0000     DIM firstnumber,secondnumber:INTEGER
      000B     DIM product:INTEGER
      0012
      0013     firstnumber:=2
      001A     secondnumber:=2
      0021
      0022     product:=firstnumber*secondnumber
      002E
      002F     PRINT
      0031     PRINT product
      0036     PRINT
      0038
      0039     END
```

THE HARD WAY

We could have made the **PROCEDURE** hardmult a lot simpler. For example, we could have written:

```
PRINT
PRINT 2*2
PRINT
```

Yes, the three lines above simply multiply two times two and print "4" on your terminal. The two extra PRINT statements print blank lines above and below the result to make the presentation look better.

Now, look at the **PROCEDURE** hardmult. We've done the same thing — multiplied two times two and printed the answer on your terminal. Why did we use so many lines?

Glad you asked — it gives us another chance to encourage you to make your programs readable. Notice that we used complete words

— words that make sense in English — when we defined the variables. We also **DIM**ensioned all our variables. We hope you will follow this practice, too. It saves a lot of memory and makes the program much easier to understand.

So why is the above example the "hardway" to solve our problem. What happens when we want to multiply four times two or 10 times 10? You guessed it. We would have to write a new procedure — or at best, edit the old one — each time we needed to multiply new numbers. Totally unsatisfactory.

EASYMULT

```
PROCEDURE easymult
     0000      PARAM firstnumber,secondnumber:INTEGER
     000B      DIM product:INTEGER
     0012
     0013      product: = firstnumber*secondnumber
     001F
     0020      PRINT
     0022      PRINT product
     0027      PRINT
     0029
     002A      END
```

THE EASY WAY

Enter the **PROCEDURE** easymult — there's always a better way to build a mousetrap. This procedure does the same job. It multiplies two numbers and prints the result. What makes it better?

The answer is almost as simple as the coding. "Easymult" is universal. It can multiply any two **INTEGER** numbers and print the correct result. It does not limit you to a set of two numbers. But, how does the procedure know which numbers you want to multiply?

It's simple. You give it two numbers. For example, you could type:

RUN EASYMULT (2,2)

You have just passed two parameters to a procedure. Your computer will respond by printing "4".

Let's try it again. Type:

"RUN EASYMULT(2,5)"

Your computer should print "10". This time, the value "2" was assigned to the variable "firstnumber". Likewise, the value "5" was assigned to the variable "secondnumber". The procedure then calculated the product, "10", and printed it.

AUTOMULT

```
PROCEDURE automult
      0000      PARAM firstnumber,secondnumber:INTEGER
      000B      DIM counter1,counter2:INTEGER
      0016      DIM answer:INTEGER
      001D
      001E      FOR counter1: = firstnumber TO secondnumber
      0030        FOR counter2: = firstnumber TO secondnumber
      0042          answer: = counter1*counter2
      004E          RUN printanswer(answer)
      0058        NEXT counter2
      0063        PRINT
      0065      NEXT counter1
      0070      END PROCEDURE printanswer
      0000      PARAM answer:INTEGER
      0007      PRINT USING "I8> "; answer
      0013      answer: = answer*2
      001E      END
```

THE AUTOMATIC WAY

You understand how to pass a parameter to a procedure from BASIC09's system mode now. But how do you teach the machine to pass parameters automatically? For the answer let's look at the procedure automult. First, enter it in your workspace and type:

RUN AUTOMULT (1,9)

I'll bet it printed a multiplication table that looked just like the one you saw when you ran the procedure demo in Chapter One. So, what's the big deal? Try:

RUN AUTOMULT (10,20)

Same format. Different numbers. Slick huh? Are you beginning to see how parameters can make your life easier. Let's look closer.

When you typed the last command, you passed the value "10" to the variable firstnumber. Likewise, the value "20" was given to secondnumber. The procedure then used these values as the beginning point and end point in a **FOR** ... **NEXT** loop. When you sent it a different set of values — different parameters if you will — the procedure started the loop in a different place and printed different numbers.

You supplied the parameters in the command line. What did the machine do for itself? Look closer.

Each time through the loop, the procedure calculated "answer". It then commanded the computer to ""**RUN** printanswer(answer)".

Each time you ran the **PROCEDURE** automult, it ran the **PROCEDURE** printanswer many times. And, every time "automult" passed a parameter to "printanswer", it had a different value. The name of that parameter, "answer", stayed the same. Only its value changed.

IT WORKS BOTH WAYS

We'll talk more about "values" and "names" shortly. But first, we'll show you that parameters can be passed in both directions. Enter BASIC09's Edit mode and add the following line to the **PROCEDURE** automult between the **RUN** statement and the **NEXT** counter2 statement.

```
PRINT USING "I5> ";answer;
```

Now **RUN** the new **PROCEDURE** automult and see what happens. If you added the new line in the right place, you'll notice that "automult" now prints twice as many columns as it did originally.

You'll see all the original "answers" plus a number of new columns that list the value of each "answer" multiplied by two. There is no line to multiply the answer by two in "automult". When were the new answers calculated?

For the answer, look closely at the **PROCEDURE** printanswer. Sure, it prints the "answer" every time "automult" asks. And, it does something else too. After it prints "answer", it assigns a new value to "answer". The new value is "answer" times two. And, that new value is passed, as a parameter, back to "automult".

You didn't notice this last bit of magic when you ran the original "automult" because the procedure just ignored the new value of "answer". Only after you added the new line, was it evident that "printanswer" had indeed changed the value of "answer".

What Value, A Name?

Why all the talk of values and names? Did you think you had picked up a Sociology textbook by accident? Rest easy. There's method to our madness. What we're really talking about here is how your computer should store your parameters. After all, we must keep them safe and sound for the procedure that's going to use them.

A parameter is really nothing more than a variable. A variable is merely a place in memory set aside to hold a value. It's value at a particular time depends on an earlier assignment statement in your program. After a statement like "answer:=4" in a procedure, the memory location set aside for the variable "answer" holds the value, "4".

Following the same logic, if your procedure contains a statement such as, "**RUN** printanswer(answer)" after the assignment above, the value "4" will be transferred to the **PROCEDURE** printanswer. In this case you have passed a parameter to another procedure "by name."

If your procedure contains the statement "**RUN** printanswer(4)", the same result appears on the terminal. But since "4" is a constant, you have passed the parameter "by value."

You may also pass a parameter "by value" by using an expression. If you type "RUN printanswer (answer$0)" or "RUN printanswer(answer*1)", the same result appears on the terminal but again you have passed the parameter "by value."

But, as my newswriting instructor so aptly commented after he read the lousy lead sentence I had just written, "Who cares?"

"You do," I said. 'And, for a very good reason."

Ponder this. If you want the procedure you are calling to return a value, you must pass the parameter to it "by name." When you pass a parameter "by name", the procedure associates the memory location of the parameter you passed with a name in a local **PARAM**eter statement. The procedure can change the value stored there and can return that value to you.

On the other hand, if you pass a parameter "by value," the procedure reacts in a completely different way. When it sees a "value" or an "expression" coming, it creates a new temporary storage area in free memory and stores the "value" there. It then uses the value where needed. When the procedure is finished, it gives the temporary storage area back to free memory. The system "forgets" that these storage locations ever existed. There is simply no way for the calling procedure to get a value back. Again, you must care.

There is a fringe benefit when you let one procedure call another by value, however. Think about it this way. Since the called procedure can't change the value of a parameter passed to it, the calling procedure has the value of its variables protected from the called procedures. That's a pretty cheap premium for insurance these days.

Before we get off the parameter kick and move on to recursion, let us leave you with a few sobering thoughts. Always make sure the parameters in your called procedure are of the same type as those in your calling procedure.

In fact, it's a good idea to make sure that the parameters in the **RUN** statement agree exactly with those expected by the **PARAM** statement in the called procedure. Otherwise, you could get some very strange results.

Also, the number of parameters must be the same. They must also be in the same order and agree with regard to size, shape, and type. Review Chapter Seven for the details of BASIC09 data types.

If you don't believe parameters in the calling and called procedures must agree and need a little mental gymnastics — sit down and write a procedure that generates an **INTEGER** in the form of an expression. Then, have it try to pass this expression to a procedure that is expecting a **BYTE** parameter.

If the called procedure is supposed to print the value of the parameter, I'll bet you'll see a lot of zeros — and, I don't mean Japanese airplanes. It's happening because the byte your called procedure is printing is being taken from the high-order byte of the two-byte **INTEGER** your calling procedure is sending.

You see, all expressions are sent to a procedure as **INTEGER** or **REAL** values. You can send **BYTE** variables to procedures till your hearts content — just remember to send them "by name."

Here's another example where parameters are passed from procedure to procedure. As you study "trig", notice that the **PROCEDURE** "display" updates the value of the parameter funcval each time it is called. Thus the variables num1 and num2 take on a new value after each pass.

TRIG

```
PROCEDURE trig
     0000
     0001      num1:=0
     0009      num2:=0
     0011
     0012      REPEAT
     0014        RUN display(num1,SIN(num1))
     0023        RUN display(num2,COS(num2))
     0032        PRINT
     0034      UNTIL num1>1
     0040
     0041      END PROCEDURE display
     0000
     0001      PARAM passed,funcval
     000A
     000B      PRINT passed; ":"; funcval,
     0019
     001A      passed:=passed+.1
     0029
     002A      END
```

RECURSION

I remember when mom and dad built their new home. They hired a contractor to put up the frame and then moved into an unfinished shell. They lived in the house while they were building it. They used that basic shell to protect them from the cold of winter while they installed the trim. Make sense? Good. You're half way down the road to understanding recursion.

When you stop to think about it for a second, you'll find there's no reason why you can't use a similar technique to build a program or procedure. Consider the **PROCEDURE** reverse.

REVERSE

```
PROCEDURE reverse
     0000
     0001     PARAM number:INTEGER
     0008
     0009     PRINT MOD(number,10);
     0012
     0013     IF INT(number/10)<>0 THEN
     0024       number:=INT(number/10)
     0031       RUN reverse(number)
     003B     ENDIF
     003D
     003E     PRINT
     0040
     0041     END
```

Reverse takes the string of decimal digits in an integer number and prints them in reverse order. Thus, the number you send it as a parameter is printed in reverse order. For Example, if you type:

```
RUN REVERSE (1234) <RETURN>
```

BASIC09 prints:

```
4321
```

Here's what happens. BASIC09's MOD function returns the remainder after an integer division. The **MOD** of 1234 is "4", so the first working line of our procedure prints that result. Next, the variable number is divided by 10.

Recursion enters the picture in the next line as the procedure "runs" itself again with the new value of number. This recursive process repeats itself until number becomes zero.

Time out for a quick warning. You won't be able to reverse an integer number larger than 32767 since that is the largest possible positive integer allowed in BASIC09.

You should also be aware that using recursion can use a lot of memory. This happens because each time a procedure calls itself, a new storage location for its local variables is created. Since there is a finite amount of storage space in your workspace, there is also a limit on the number of times a procedure can call itself before running out of memory.

Since we've gotten you interested in recursion, let's give you a few more examples to run and study.

HANOI

```
PROCEDURE hanoi
     0000
     0001     REM from Basic09 Manual
     0013     REM move n discs in Tower of Hanoi Game
     0039     REM see BYTE, Oct. 1980, Pg. 279
     0058
     0059     PARAM number:INTEGER; from,overto,other:STRING[8]
     0073
     0074     IF number=1 THEN
     0080       PRINT "move /"; number; " from "; from; " to "; overto
     00A6     ELSE
     00AA       RUN hanoi(number-1,from,other,overto)
     00C5       PRINT "Move #"; number; " from "; from; " to "; overto
     00EB       RUN hanoi(number-1,other,overto,from)
     0106     ENDIF
     0108
     0109     END
     010B
```

This procedure is a version of the famous Towers of Hanoi. The object of the game is to move the rings from one post to another using the minimum number of moves. Notice that each time the procedure runs, it runs itself again with a new value of number. This happens until the value of number is "1". When that happens, the problem has been solved.

Here's a hint. Notice that the **PROCEDURE** hanoi requires four parameters when it is called. Notice further that one of these parameters is an integer number and the rest are strings. BASIC09 requires that you place quotes around your strings when you enter parameters. The command line to run "Hanoi", after you have loaded it into your workspace is:

```
RUN HANOI (8,"Dale's","Esther's","Michele's")
```

SUMMARY

In this chapter you've been introduced to several powerful concepts. You've learned more about running BASIC09 procedures from a command line while operating in the system mode.

You've been introduced to parameters and discovered how to use them to pass data to a procedure. In fact you've even learned how to get information back from a procedure.

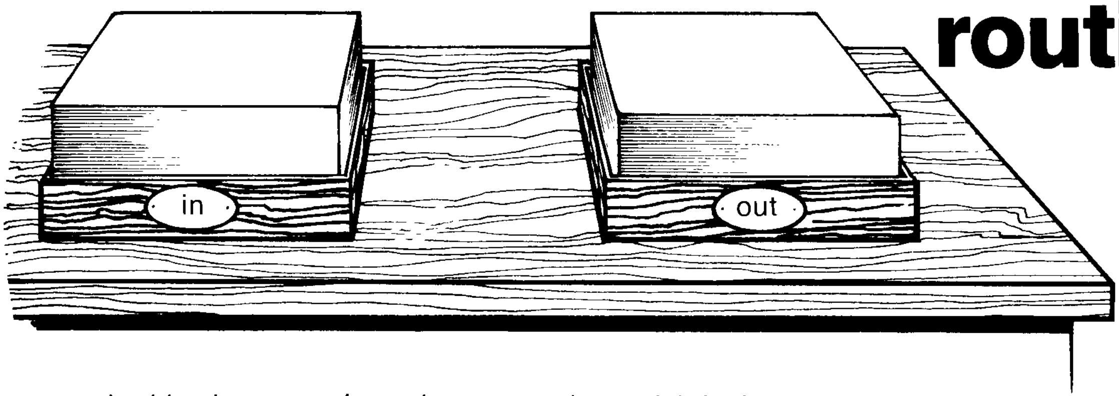
You've learned how to write procedures that run other procedures automatically.

And, you've been introduced to recursion as we showed you procedures that run themselves over and over again.

In Chapter 13 you'll learn about using 6809 machine language subroutines with BASIC09. May your pit stops always be short and your Mercedes on course.

CHAPTER THIRTEEN

using machine language routines



In this chapter we're going to sneak a quick look at a subject that may be alien to you — 6809 machine language subroutines. This is really a topic for advanced programmers. The truth is that you may write programs in BASIC09 for years and you may never need to tangle with machine language.

That's because BASIC09 is so powerful that you may never need machine code. Yet, there are times — especially when you are running real time applications — that you will want its speed. We'll try to remove some of the mystery.

One warning before we get started: the subject of how to program in machine language is way beyond the scope of this book. If you don't already know how, either get a book on the subject or skip this chapter altogether.

STACKS

To understand how BASIC09 talks to your machine language routine you will need to understand the concept of stacks. BASIC09 sends its parameters to your machine language routine on a stack and expects to find the results there when your subroutine returns control.

Here's an analogy. An executive uses an "in-basket" to hold incoming correspondence. When his secretary brings in letters and memos, she places them in the basket. After he has answered several telephone calls and attended two or three meetings the basket usually contains quite a stack of paper.

If the manager works like most of us he will take a letter off the top of the stack and work on it first. This works fine — unless you're an eager beaver and delivered your report first. If you are, your hard work will be buried at the bottom of the stack. You might have been the first to deliver, but you'll be the last to be read.

Our manager is using what is known in the computer world as a **LIFO** — a last in, first out stack. Eager beavers don't think much of them. **LIFO**'s are common in many computer languages and are an easy way to get you to think about stacks.

Now, let's pretend that each piece of paper in the manager's in-basket is a memory location in your computer. The first item is stored in memory location number one, the next one in number two and so forth.

If you just knew where the first item was located in memory, you would know the location of every piece of information. This is where the magic of your 6809 microprocessor enters the plot. The 6809 uses one of its registers to point to the stack — memory location number one as we called it earlier. It's called the stack pointer.

BASIC09 uses a stack to transfer data to and from its own procedures and machine language routines. This stack is very organized and always holds the same information at the same position in the stack.

Your machine language routine uses this 6809 register — the hardware stack pointer — to find the information or parameters BASIC09 puts in the stack for it. It also uses this register to send the results back to BASIC09. Each specific piece of information BASIC09 sends to your routine is stored at a fixed distance from the address held by this register. The fixed distance is known as an offset.

The address pointed to by the stack register has an offset of zero. The next address has an offset of one, the next two, etc. The stack pointer and the offsets make up what is known as a stack frame. Here's what BASIC09's stack frame looks like.

BASIC09 STACK FRAME

```
Size of Last Parameter
Address of Last Parameter
.
.
.
10,S Size of Second Parameter
8,S Address of Second Parameter
6,S Size of First Parameter
4,S Address of First Parameter
2,S Number of Parameters being passed
0,S Return Address of BASIC09 routine
```

The address held in the 6809's hardware stack pointer when your routine is called is at the bottom of stack frame shown above. In fact the coding, "0,S" is a way of expressing an address in 6809 assembly language. In English, "0,S" means the memory location pointed to by the 6809 S-register — or stack pointer. In 6809 assembly language this is known as "indexed" addressing.

Let's take it one step farther. The "2,S" refers to the memory location two bytes higher than the address held in the S-register. For example if the S-register holds a value of $A000, then 2,S would be $A002 ... 4,S would represent $A004, etc.

LET'S ADD IT ALL UP

Since the BASIC09 Tour Guide is concerned mainly with a higher level language, we'll only give you a short and simple assembly language programming example here. Hopefully it will whet your appetite and give you a starting point for your study of this new and exciting world.

As an example, we'll show you a routine to add two **INTEGER** numbers. It is written in 6809 assembly language. Granted, there is no need for a routine like this with BASIC09, but we need a place to start.

```
* Add two INTEGER numbers
* Called from BASIC09 by:
*
* RUN addtwo(numone,numtwo)
*          * Where numone and numtwo are INTEGER parameters
*
Type      SET    SBRTN+OBJCT
Revs      SET    REENT+1
MOD addend,addnam,type,revs,addent,0
addnam     FCS /addtwo/
addent    LDD [4,S]        get numone
          ADDD [8,S]       add to numtwo
          STD [4,S]        return result in numone
          CLRB             indicate no error
          RTS              return to BASIC09
          EMOD
addend    EQU *
          END
```

How does it all work? First, the **RUN** statement will look for a module named "addtwo" in BASIC09's workspace. Since the routine is a 6809 machine code subroutine, it won't find it there. Machine code is always loaded into system memory.

When **RUN** doesn't find "addtwo" in BASIC09's workspace, it will search OS-9's module directory to see if there is a module in memory with that name. If "addtwo" has been loaded into memory, RUN will find it and check to see what type of code it contains.

If the module contains 6809 machine code — like "addtwo" — **RUN** makes a subroutine call to the module's entry address. When the return from subroutine instruction, **RTS,** is encountered, control returns to BASIC09.

If **RUN** doesn't find "addtwo" in system memory, it still won't give up. It will attempt to load the module from a file by the same name in the current execution directory. You almost have more chances than a cat.

To understand what happens when you type **RUN** addtwo(numone,numtwo) from BASIC09's system mode, we must take a close look at the stack frame introduced above.

Let's pretend that your calling procedure sets numone equal to "2" and numtwo equal to "4" before it **RUN**s addtwo. Your stack frame should look like this.

LOCATION	**CONTENTS**	
10,S	**0002**	**length of two byte INTEGER**
8,S	**address of numtwo**	
6,S	**0002**	**length of two byte INTEGER**
4,S	**address of numone**	
2,S	**0002**	**we have two parameters**
0,S	**RETURN address**	**BASIC09's reentry address**

Before BASIC09 turns control over to the subroutine "addtwo" by executing a jump to subroutine instruction it loads the 6809 stack register with the address of the stack frame above. The two byte value of the **INTEGER** variable numone will have been stored at some address in memory. That address will then be stored four bytes above the stack pointer in the stack frame — at "4,S". Likewise, the address where the variable numtwo is stored will be placed in the memory location eight bytes above the stack pointer.

BASIC09 puts the return address at the memory location pointed to by the stack pointer and puts the length of the two variables in the memory locations six and 10 bytes above the stack pointers address. Since the variables numone and numtwo are **INTEGER** variables, they have a length of two bytes. Had they been **REAL** variables, they would have had a length of five.

When the machine code gets control it loads the 6809's D-register with the value stored at the memory location pointed to by the value stored at the location, "4,S". The brackets in the code, "[4,S]", tell the assembler that you want to load the D-register with this data instead of the value stored at 4,S itself. The brackets indicate the 6809's indexed, indirect addressing mode.

The routine then adds the value it loaded into the D-register with the value pointed to by the data at 8,S and stores it in the address pointed to by the data at 4,S — or numone. Before returning to BASIC09, "addtwo" clears the 6809 B-register to indicate there were no errors encountered during the operation.

After the **RUN** statement, it you **PRINT** numone, you should get a result of "6". This result happens because your machine language routine has stored its result in the first parameter — the one your BASIC09 calling procedure named "numone".

SUMMARY

You should have enough ammunition to get you started linking BASIC09 procedures to assembly language routines now. If you're already a seasoned assembly language programmer, you should have no problems. If not, you should consult one of the many fine 6809 assembly language primers now on the market. You'll also find several examples in Microware's BASIC09 Reference Manual.

In Chapter 14 we'll present some interesting programs for you to study and for your amusement.

CHAPTER FOURTEEN

example programs

SAMPLE PROGRAM ONE—FINANCES

```
PROCEDURE Finance
  DIM Selection:BYTE

  LOOP
    RUN Clearscreen

    PRINT "F i n a n c i a l   C a l c u l a t i o n s"
    PRINT "= = = = = = = = = = = = = = = = = = = = = = = ="
    PRINT
    PRINT "1 — Investments"
    PRINT "2 — Depreciation"
    PRINT "3 — Loans"
    PRINT "0 — I Quit!"

    RUN Prompt(Selection)

  EXITIF Selection=0 THEN
    RUN Clearscreen
    PRINT
    PRINT "Thank You for letting me help with your finances."
    PRINT
  ENDEXIT

    IF Selection=1 THEN RUN Invest
    ELSE
      IF Selection=2 THEN RUN Depreciate
      ELSE
        IF Selection=3 THEN RUN Loan
        ELSE
          RUN EntryError
        ENDIF
      ENDIF
    ENDIF
  ENDLOOP
  END
```

```
PROCEDURE ClearScreen
    DIM clearscreen:STRING[1]
    clearscreen:=CHR$($1C)
    PRINT clearscreen
    END

PROCEDURE Printline
    DIM line:STRING[51]
      line:=" --------------------------------------------------- "
    PRINT line
    END

PROCEDURE Prompt
    PARAM Selection:BYTE
    DIM Prompt1:STRING[32]
    DIM Prompt2:STRING[32]

    Prompt1:="Please select one of the options"
    Prompt2:="by typing the proper number"

    PRINT
    PRINT Prompt1
    PRINT Prompt2;
    INPUT Selection

    END



PROCEDURE YesOrNo
    PARAM Answer:STRING[1]
    DIM response:STRING[1]

    REPEAT

      PRINT
      INPUT "Would you like to make another calculation (Y) or (N)? "
      ,response

      RUN clearscreen

    UNTIL response="Y" OR response="y" OR response="N" OR response ="n"

    Answer:=response
    END



PROCEDURE EntryError
    DIM response:STRING[1]
    PRINT
    PRINT "Your answer is not valid."
    PRINT "Hit any key to continue. ";
    GET #0,response
    PRINT
    END
```

```
PROCEDURE MakeItPretty
    PARAM Value:REAL
    PRINT USING "T40,R12.2> ",Value
    END


PROCEDURE DisplayValue
    PARAM Investment,Interest,Value:REAL

    PRINT
    RUN Printline

    PRINT "Amount invested:  ";
    RUN MakeItPretty(Investment)

    PRINT "Value of accumulated interest: ";
    RUN MakeItPretty(Interest)

    PRINT "Total Value of your investment: ";
    RUN MakeItPretty(Value)

    RUN Printline
    PRINT
    END


PROCEDURE Invest
    DIM Selection:BYTE

    LOOP
      RUN Clearscreen
      PRINT "I n v e s t m e n t   C a l c u l a t i o n s"
      PRINT "= = = = = = = = = = = = = = = = = = = = = = = = ="
      PRINT
      PRINT "1 — Future value of a one-time investment"
      PRINT "2 — Future value of regular deposits"
      PRINT "3 — Regular deposits required to create a desired value"
      PRINT "0 — That's All Folks"

      RUN prompt(Selection)

    EXITIF Selection=0 THEN
    ENDEXIT

      IF Selection=1 THEN RUN onetimefuture
      ELSE
        IF Selection=2 THEN RUN futureregdeposit
        ELSE
        IF Selection=3 THEN RUN DepositsRequired
          ELSE
            RUN EntryError
          ENDIF
        ENDIF
      ENDIF
    ENDLOOP
    END
```

```
PROCEDURE Depreciate
    DIM Selection:BYTE

    LOOP
      RUN ClearScreen
      PRINT "D e p r e c i a t i o n   C a l c u l a t i o n s"
      PRINT "= = = = = = = = = = = = = = = = = = = = = = = = = ="
      PRINT
      PRINT "1 — Annual Depreciation Rate"
      PRINT "2 — Amount of Depreciation"
      PRINT "3 — Salvage Value"
      PRINT "0 — Return to Main Menu"

      RUN prompt(Selection)

    EXITIF Selection=0 THEN
    ENDEXIT

    IF Selection=1 THEN RUN AnnualRate
    ELSE
      IF Selection=2 THEN RUN DepreciationAmount
      ELSE
        IF Selection=3 THEN RUN Salvage
        ELSE
          RUN EntryError
        ENDIF
      ENDIF
    ENDIF
  ENDLOOP
  END



PROCEDURE Loan
      DIM Selection:BYTE

    LOOP
      RUN ClearScreen
      PRINT "L o a n   C a l c u l a t i o n s"
      PRINT "= = = = = = = = = = = = = = = = = ="
      PRINT
      PRINT "1 — Regular Payments on a Loan"
      PRINT "2 — Last Payment of a Loan"
      PRINT "3 — Term of a Loan"
      PRINT "4 — Remaining Balance on a Loan"
      PRINT "5 — Cost of Borrowing"
      PRINT "0 — Who Needs the 'Loan Arranger'"

      RUN prompt(Selection)

    EXITIF Selection=0 THEN
    ENDEXIT
    IF Selection = 1 THEN RUN RegularPay
    ELSE
      IF Selection = 2 THEN RUN LastPay
      ELSE
        IF Selection = 3 THEN RUN LoanTerm
        ELSE
          IF Selection = 4 THEN RUN Balance
          ELSE
            IF Selection = 5 THEN RUN BorrowingCost
            ELSE
              RUN EntryError
            ENDIF
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  ENDLOOP
```

```
PROCEDURE OneTimeFuture
    DIM Investment,Rate,value,interest,years:REAL
    DIM months,periods:INTEGER
    DIM Response:STRING[1]

    RUN ClearScreen

    PRINT "F u t u r e   V a l u e   o f   a   O n e - T i m e   I n v e s t m e n t"
    PRINT "= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = ="
    PRINT

    INPUT "What is your initial investment? ",Investment
    INPUT "What is the nominal interest rate? ",Rate
    INPUT "Term of your investment (Years,Months)? ",years,months
    INPUT "How many compounding periods each year? ",periods


    Rate: = Rate/periods/100
    years: = (12*years + months)/12
    value: = Investment*(1 + Rate)↑(periods*years)
    value: = INT(value*100 + .5)/100
    interest: = value-Investment

    RUN DisplayValue(Investment,interest,value)
    RUN YesOrNo(Response)

    IF Response = "Y" OR Response = "y" THEN
      RUN OneTimeFuture
    ENDIF
    END
```

```
PROCEDURE futureregdeposit
    DIM Deposit,Rate,Years,Value,Investment,Interest:REAL
    DIM Months,Periods:INTEGER
    DIM Response:STRING[1]

    RUN ClearScreen
    PRINT "F u t u r e   V a l u e   o f   R e g u l a r   D e p o s i t s"
    PRINT "= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = ="
    PRINT

    INPUT "How much is each Deposit? ",Deposit
    INPUT "What is the nominal interest rate? ",Rate
    INPUT "Term of investment (Years, Months)? ",Years,Months
    INPUT "How many deposits are you making per year? ",Periods

    Rate:=Rate/Periods/100
    Years:=(12*Years+Months)/12
    Value:=Deposit*((1+Rate)↑(Periods*Years)-1)/Rate
    Value:=INT(Value*100+.5)/100
    Investment:=Deposit*Years*Periods
    Interest:=Value-Investment

    RUN DisplayValue(Investment,Interest,Value)

    RUN YesOrNo(Response)

    IF Response="Y" OR Response="y" THEN
      RUN futureregdeposit
    ENDIF

    END




PROCEDURE DepositsRequired
    DIM Value,Rate,Years,Periods,Deposit,Interest,Investment:REAL
    DIM Response:STRING[1]

    RUN ClearScreen

    PRINT "R e q u i r e d   R e g u l a r   D e p o s i t s"
    PRINT "= = = = = = = = = = = = = = = = = = = = = = = = = ="
    PRINT

    INPUT "What is the final value you desire? ",Value
    INPUT "What is the nominal interest rate? ",Rate
    INPUT "Term of investment (Years, Months)? ",Years,Months
    INPUT "How many deposits are you making a year? ",Periods
    Rate:=Rate/Periods/100
    Years:=(12*Years+Months)/12
    Deposit:=Value*Rate/((1+Rate)↑(Periods*Years)-1)
    Investment:=Deposit*Years*Periods
    Interest:=Value-Investment

    PRINT
    RUN Printline
    PRINT "Amount of required regular Deposits: ";
    RUN MakeItPretty(Deposit)
    RUN Printline
    RUN YesOrNo(Response)

    IF Response="Y" OR Response="y" THEN
      RUN DepositsRequired
    ENDIF
    END



PROCEDURE AnnualRate
    DIM pricepaid,pricesold,years,rate:REAL
    DIM months:INTEGER
    DIM Response:STRING[1]

    RUN ClearScreen

    PRINT "D e p r e c i a t i o n   A m o u n t"
    PRINT "= = = = = = = = = = = = = = = = = = = ="
    PRINT

    INPUT "How much did you pay for the item? ",pricepaid
    INPUT "How much can you sell it for? ",pricesold
    INPUT "Depreciation Term (Years, Months)? ",years,months

    years:=(12*years+months)/12
    rate:=1-(pricesold/pricepaid)↑(1/years)
    rate:=INT(rate*100+.5)

    PRINT
    RUN Printline
    PRINT "Depreciation Rate (Percent): ";
    RUN MakeItPretty(rate)
    RUN Printline
    RUN YesOrNo(Response)

    IF Response="Y" OR Response="y" THEN
      RUN AnnualRate
    ENDIF
    END
```

```
PROCEDURE DepreciationAmount
    DIM PricePaid,Rate,Years,Depreciation:REAL
    DIM months:INTEGER
    DIM Response:STRING[1]

    RUN ClearScreen

    PRINT "D e p r e c i a t i o n   A m o u n t "
    PRINT "= = = = = = = = = = = = = = = = = = = = "
    PRINT

    INPUT "How much did you pay for the item? ",PricePaid
    INPUT "What is the Depreciation Rate? ",Rate
    INPUT "What is the year of Depreciation? ",Years

    Rate:=Rate/100
    Depreciation:=PricePaid*Rate*(1-Rate)↑(Years-1)

    PRINT
    RUN Printline
    PRINT "The amount of depreciation now is: ";
    RUN MakeItPretty(Depreciation)
    RUN Printline
    RUN YesOrNo(Response)

    IF Response="Y" OR Response="y" THEN
      RUN DepreciationAmount
    ENDIF
    END




PROCEDURE Salvage
    DIM PricePaid,Rate,SalvageValue:REAL
    DIM Months:INTEGER
    DIM Response:STRING[1]

    RUN ClearScreen

    PRINT "S a l v a g e   V a l u e"
    PRINT "= = = = = = = = = = = = = = "
    PRINT

    INPUT "How much did you pay for the item? ",PricePaid
    INPUT "What is the depreciation rate? ",Rate
    INPUT "Depreciation Term (Years, Months)? ",Years,Months

    Rate:=Rate/100
    SalvageValue:=PricePaid*(1-Rate)↑Years
    PRINT
    RUN Printline
    PRINT "The salvage value is: ";
    RUN MakeItPretty(SalvageValue)
    RUN Printline
    RUN YesOrNO(Response)

    IF Response = "Y" OR Response = "y" THEN
      RUN Salvage
    ENDIF
    END
```

```
PROCEDURE RegularPay
    DIM amountborrowed,years,rate,Holder,Payment:REAL
    DIM months,periods:INTEGER
    DIM Response:STRING[1]

    RUN ClearScreen

    PRINT "R e g u l a r   P a y m e n t   o n   a   L o a n"
    PRINT "= = = = = = = = = = = = = = = = = = = = = = = = = = ="
    PRINT

    INPUT "How much do you want to borrow? ",amountborrowed
    INPUT "What is the term of the loan (Years, Months)? ",years ,months
    INPUT "What is the annual interest rate? ",rate
    INPUT "How many payments will you make per year? ",periods

    rate: = rate/periods/100
    years: = (years*12 + months)/12
    Holder: = 1/(1 + rate)↑(periods*years)
    Payment: = amountborrowed*rate/(1-Holder)

    PRINT
    RUN printline
    PRINT "Your regular payment will be: ";
    RUN MakeItPretty(Payment)
    RUN printline
    RUN yesorno(Response)

    IF Response = "Y" OR Response = "y" THEN
      RUN RegularPay
    ENDIF

    END
```

```
PROCEDURE LastPay
    DIM borrowed,years,rate,regularpay,interestpayment,payonprincipal
    ,payholder:REAL
    DIM months,periods,numberpayments,count:INTEGER
    DIM Response:STRING[1]

    RUN ClearScreen

    PRINT "L a s t   P a y m e n t   o n   a   L o a n"
    PRINT "========================"
    PRINT

    INPUT "How much are you going to borrow? ",borrowed
    INPUT "What is the term of the loan (years, months)? ",years
    ,months
    INPUT "What is the annual interest rate? ",rate
    INPUT "How many payments will you make per year? ",periods
    INPUT "How much is your regular payment? ",regularpay

    rate:=rate/periods/100
    years:=(years*12+months)/12
    numberpayments:=periods*years

    FOR count:=1 TO numberpayments
      interestpayment:=INT(borrowed*rate*100+.5)/100
      payonprincipal:=regularpay-interestpayment
      borrowed:=borrowed-payonprincipal
    NEXT count

    payholder:=regularpay+borrowed

    PRINT
    RUN Printline
    PRINT "Your last payment will be: ";
    RUN MakeItPretty(payholder)
    RUN Printline
    RUN YesOrNo(Response)

    IF Response="Y" OR Response="y" THEN
      RUN LastPay
    ENDIF
    END
```

```
PROCEDURE LoanTerm
    DIM principal,regularpay,rate,term1,term2,term:REAL
    DIM periods,months,years:INTEGER
    DIM response:STRING[1]

    RUN ClearScreen

    PRINT "T e r m   o f   a   L o a n"
    PRINT "=============== "
    PRINT

    INPUT "How much do you hope to borrow? ",principal
    INPUT "How much is your regular payment? ",regularpay
    INPUT "What is the annual interest rate? ",rate
    INPUT "How many payments will you make a year? ",periods

    rate:=rate/periods/100
    term1:=1-principal*rate/regularpay
    term2:=1+rate
    term:=-(LOG(term1)/LOG(term2))/periods
    months:=INT(term*12)
    years:=INT(months/12)
    months:=months-years*12

    PRINT
    RUN Printline
    PRINT "The term of your loan would be: ";
    PRINT USING "I4>",years;
    PRINT " years";
    PRINT USING "I4>"; months;
    PRINT " months."

    RUN Printline
    RUN YesOrNo(response)

    IF response="Y" OR response="y" THEN
      RUN LoanTerm
    ENDIF
    END
```

```
PROCEDURE balance
    DIM principal,regularpay,rate,rate1,rate2:REAL
    DIM count,paymentsperyear,paymentsmade:INTEGER
    DIM response:STRING[1]

    RUN ClearScreen
    PRINT "R e m a i n i n g   b a l a n c e   o n   a   L o a n"
    PRINT "=============================="
    PRINT
    INPUT "How much did you borrow? ",principal
    INPUT "How much is your regular payment? ",regularpay
    INPUT "What is the annual interest rate? ",rate
    INPUT "How many payments do you make a year? ",paymentsperyear
    INPUT "How many payments have you made? ",paymentsmade

    rate:=rate/paymentsperyear/100

    FOR count:=1 TO paymentsmade
      rate1:=INT(principal*rate*100+.5)/100
      rate2:=regularpay-rate1
      principal:=principal-rate2
    NEXT count

    PRINT
    RUN Printline
    PRINT "Your remaining balance is: ";
    RUN MakeItPretty(principal)
    RUN Printline
    RUN YesOrNo(response)

    IF response="Y" OR response="y" THEN
      RUN balance
    ENDIF
    END
```

```
PROCEDURE BorrowingCost
      DIM pay4,pay5,cost,rate1,rate2,costofborrowing:REAL
      DIM principal,years,rate,pay:REAL
      DIM numofpayments,months,payperyear,count:INTEGER
      DIM response:STRING[1]
```

```
RUN ClearScreen

PRINT "C o s t   o f   B o r r o w i n g"
PRINT "= = = = = = = = = = = = = = = = = = ="
PRINT

INPUT "How much do you hope to borrow? ",principal
INPUT "What is the term of the loan (Years, Months)? ",years
,months
INPUT "What is the annual interest rate? ",rate
INPUT "How many payments will you be making each year? ",payperyear

  rate:=rate/payperyear/100
  years:=(years*12+months)/12
  pay3:=1/(1+rate)↑(payperyear*years)
  pay4:=principal*rate/(1-pay3)
  pay4:=INT(pay4*100+.5)/100
  pay5:=principal
  cost:=0
  numofpayments:=payperyear*years

  FOR count:=1 TO numofpayments
    rate1:=INT(pay5*rate*100+.5)/100
    rate2:=pay4-rate1
    pay5:=pay5-rate2
    cost:=cost+pay4
  NEXT count

  cost:=cost+pay5
  costofborrowing:=cost-principal

  PRINT
  RUN Printline
  PRINT "Regular Payments: ";
  RUN MakeitPretty(pay4)
  PRINT "Total Payments: ";
  RUN MakeitPretty(cost)
  PRINT "Cost of Borrowing: ";
  RUN MakeitPretty(costofborrowing)
  RUN Printline
  PRINT

RUN YesOrNo(response)

  IF response="Y" OR response="y" THEN
    RUN BorrowingCost
  ENDIF
  END
```

SAMPLE PROGRAM TWO—BLACKJACK GAME

```
PROCEDURE blkjak
    TYPE c=cards(52),cardnumber:INTEGER
    DIM deck:c
    TYPE dealer=ccount,acecount,showcard:INTEGER; bjack:BOOLEAN;
      holecard$:STRING[20]
    TYPE player=ccnt,acecnt,card1,card2,doublecount(3):INTEGER;
      bj:BOOLEAN; name$,card$(3):STRING[20]
    DIM d:dealer; p:player
    DIM balance,startbalance,wager,i,temp,bet(3):INTEGER
    DIM card:STRING[20]; reply:STRING[1]
    DIM score:REAL

    temp=RND(-(VAL(RIGHT$(DATE$,2))))
    FOR i=1 TO 52
      deck.cards(i)=i-1
    NEXT i
    deck.cardnumber=52
    PRINT
    PRINT "***** Las Vegas Blackjack *****"
    PRINT
    REPEAT
    INPUT "What is your name? ",p.name$
      p.name$=TRIM$(p.name$)
    UNTIL p.name$<>""
    IF LEFT$(p.name$,1)>="a" AND LEFT$(p.name$,1)<="z" THEN
      p.name$=CHR$(ASC(p.name$)-$20)+MID$(p.name$,2,255)
    ENDIF

    LOOP
      INPUT "Do you want a list of the rules (Y/N)? ",reply

      IF reply<>"N" THEN
        PRINT
        PRINT "The rules of this casino are as follows..."
        PRINT " > BlackJack pays 3 to 2."
        PRINT " > Pairs may be split."
        PRINT " > You only get one card on each split ace."
        PRINT " > You may double down on a split pair."
        PRINT " > You do not lose a tie hand."
        PRINT " > Dealer hits 16, stands on all 17's."
        PRINT " > Wagers are 1 to 500 dollars."
        PRINT " > Type Return to draw a card"
        PRINT " > Type S        to stand with your present cards."
        PRINT " > Type D        to go down for doubles."
        PRINT " > Type X        to split your pairs."
        PRINT " > Bet 0 dollars to terminate a game."
        PRINT "* Good Luck *"
      ENDIF
```

```
PRINT
PRINT "This game started at "; RIGHT$(DATE$,8)
startbalance = RND(1000) + 50
balance = startbalance
PRINT "Your bankroll for this game is "; balance; " dollars."
LOOP
  IF deck.cardnumber> =49 THEN
    RUN shuffle(deck)
  ENDIF
  PRINT
  LOOP
    INPUT "Wager? ",wager
  EXITIF wager> =0 AND wager< =500 THEN ENDEXIT
    PRINT "House limits are between 1 and 500 dollars."
  ENDLOOP
  IF wager< >0 THEN
    p.ccnt = 0 \ p.acecnt = 0
    d.ccount = 0 \ d.acecount = 0
    RUN Draw(deck,p.ccnt,p.acecnt,p.card1,p.card$(1))
    RUN Draw(deck,d.ccount,d.acecount,d.showcard,card)
    PRINT "Dealer shows:"; TAB(20); card
    RUN Draw(deck,p.ccnt,p.acecnt,p.card2,p.card$(2))
    PRINT p.name$; " has: "; TAB(20); p.card$(1); ", "; p.card$(2)
    RUN Draw(deck,d.ccount,d.acecount,temp,d.holecard$)
    p.bj = p.ccnt = 21
    d.bjack = d.ccount = 21
    IF d.showcard = 11 THEN
      RUN ibet(d,p,balance,wager)
    ENDIF
    IF d.bjack THEN
      PRINT "Dealer has BlackJack."
    ELSE
      LOOP
        INPUT reply
      EXITIF p.card1 = p.card2 AND reply = "X" THEN ENDEXIT
      EXITIF SUBSTR(reply,"SDX")< >0 THEN ENDEXIT
        PRINT "Invalid request—retry."
      ENDLOOP
      IF reply = "" OR reply = "D" THEN
        RUN playermove(deck,p,reply,wager,p.ccnt)
        p.bj = FALSE
      ENDIF
      IF reply = "X" THEN
        FOR i = 1 TO 2
          bet(i) = wager
          p.doublecount(i) = p.card1
          IF p.card1 = 11 THEN p.acecnt = 1
          ENDIF
          RUN Draw(deck,p.doublecount(i),p.acecnt,temp,card)
          PRINT "Hand #"; i; "           "; p.card$(i); ", "; card
          IF p.card1< >11 THEN
            LOOP
```

```
                    INPUT reply
                  EXITIF SUBSTR(reply,"SD")< >0 THEN ENDEXIT
                    PRINT "Invalid request—retry."
                  ENDLOOP
                ENDIF
                RUN playermove(deck,p,reply,bet(i),p.doublecount(i))
              NEXT i
              RUN dealermove(deck,d,p)
              FOR i=1 TO 2
                PRINT "Hand #"; i
                RUN winloss(d,p,p.doublecount(i),bet(i),balance)
              NEXT i
            ENDIF
          ENDIF
          IF reply< >"X" THEN
            IF p.bj THEN
              PRINT p.name$; " has BlackJack."
              wager=1.5*wager
            ENDIF
            RUN dealermove(deck,d,p)
            RUN winloss(d,p,p.ccnt,wager,balance)
          ENDIF
        ENDIF
      EXITIF wager=0 THEN ENDEXIT
      ENDLOOP
      IF balance<0 THEN
        PRINT p.name$; " owes the casino "; -(balance); " dollars."
      ENDIF
      score=balance-startbalance
      RUN topscores(" Blackjack ","won",score,"dollars",score,TRUE)
      INPUT "New game (Y/N)? ",reply
    EXITIF reply="N" THEN ENDEXIT
    ENDLOOP
    END

PROCEDURE dealermove
    TYPE c=cards(52),cardnumber:INTEGER
    TYPE dealer=ccount,acecount,showcard:INTEGER; bjack:BOOLEAN;
      holecard$:STRING[20]
    TYPE player=ccnt,acecnt,card1,card2,doublecount(3):INTEGER; bj
      :BOOLEAN; name$,card$(3):STRING[20]
    PARAM deck:c; d:dealer; p:player
    DIM temp:INTEGER; card:STRING[20]

    PRINT "Dealer hole card "; d.holecard$
    WHILE d.ccount<17 AND NOT(p.bj) AND p.ccnt< =21 DO
      RUN Draw(deck,d.ccount,d.acecount,temp,card)
      PRINT "Dealer hits "; card
    ENDWHILE
    IF d.ccount>21 THEN
      PRINT "Dealer Busts."
    ENDIF
    END
```

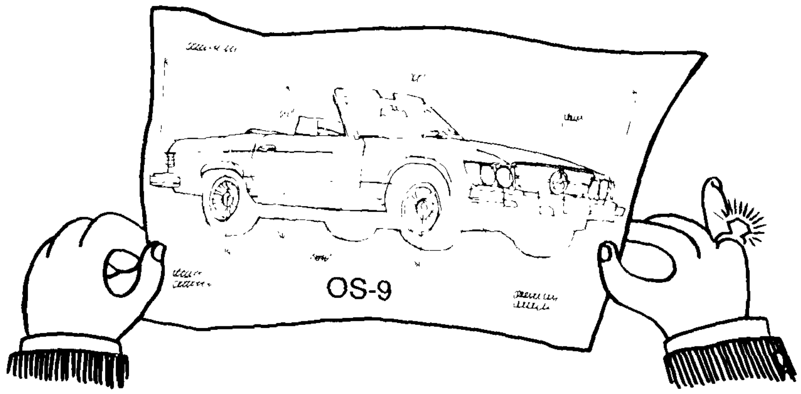
```
PROCEDURE winloss
    TYPE dealer=ccount,acecount,showcard:INTEGER; bjack:BOOLEAN;
      holecard$:STRING[20]
    TYPE player=ccnt,acecnt,card1,card2,doublecount(3):INTEGER; bj
      :BOOLEAN; name$,card$(3):STRING[20]
    PARAM d:dealer; p:player; count,wager,balance:INTEGER
    IF NOT(p.bj) AND NOT(d.bjack) AND count< =21 AND d.ccount< =21 THEN
      PRINT p.name$; "'s count= "; count; ", Dealer's count5"; d.ccount
    ENDIF
    IF d.bjack AND NOT(p.bj) OR count>21 OR count<d.ccount AND d.ccount<22
THEN
      balance=balance-wager
      PRINT "Dealer wins this hand — Bankroll= "; balance; " dollars."
    ELSE
      IF count=d.ccount AND count<22 THEN
        PRINT "This hand is a tie — Bankroll= "; balance; " dollars."
      ELSE
        balance=balance+wager
        PRINT p.name$; " wins this hand — Bankroll= "; balance; " dollars."
      ENDIF
    ENDIF
    END

PROCEDURE playermove
    TYPE c=cards(52),cardnumber:INTEGER
    TYPE player=ccnt,acecnt,card1,card2,doublecount(3):INTEGER; bj
      :BOOLEAN; name$,card$(3):STRING[20]
    PARAM deck:c; p:player; reply:STRING[1]; wager,count:INTEGER
    DIM temp:INTEGER; card:STRING[20]
    IF reply="D" THEN
      RUN Draw(deck,count,p.acecnt,temp,card)
      wager=2*wager
      PRINT p.name$; " doubles down "; card
      IF count>22 THEN PRINT p.name$; " busts."
      ENDIF
      END
    ENDIF
    REPEAT
      RUN Draw(deck,count,p.acecnt,temp,card)
      PRINT p.name$; " hits "; card
      IF count>21 THEN
        PRINT p.name$; " Busts."
        END
      ENDIF
      LOOP
        INPUT reply
      EXITIF SUBSTR(reply,"S")< >0 THEN ENDEXIT
        PRINT "Invalid request—retry."
      ENDLOOP
    UNTIL reply="S"
    END
```

```
PROCEDURE ibet
    TYPE dealer=ccount,acecount,showcard:INTEGER; bjack:BOOLEAN;
      holecard$:STRING[20]
    TYPE player=ccnt,acecnt,card1,card2,doublecount(3):INTEGER; bj
      :BOOLEAN; name$,card$(3):STRING[20]
    PARAM d:dealer; p:player; balance,wager:INTEGER
    DIM reply:STRING[1]

    LOOP
      INPUT "Insurance bet (Y or N)? ",reply
    EXITIF SUBSTR(reply,"YN")<>0 THEN ENDEXIT
    ENDLOOP
    IF d.bjack THEN
      IF reply="Y" THEN
        balance=balance+wager
        PRINT p.name$; " wins insurance bet of "; wager; " dollars."
      ENDIF
    ELSE
      IF reply="Y" THEN
        balance=balance-.5*wager
        PRINT "Dealer does not have BlackJack."
        PRINT p.name$; " loses insurance bet of "; .5*wager; " dollars."
      ENDIF
      PRINT "Hand continues; enter requests as usual."
    ENDIF
    END




PROCEDURE draw
    TYPE d=cards(52),cardnumber:INTEGER
    PARAM deck:d; ccount,acecount,cardvalue:INTEGER
    PARAM card$:STRING[20]
    DIM r,s:INTEGER

    IF deck.cardnumber>=52 THEN
      RUN SHUFFLE(deck)
    ENDIF
    cardvalue=deck.cards(deck.cardnumber)
    deck.cardnumber=deck.cardnumber+1
    s=cardvalue/13
    r=MOD(cardvalue,13)
    cardvalue=r+1
    IF cardvalue>10 THEN cardvalue=10
    ENDIF
    IF cardvalue = 1 THEN cardvalue = 11
    ENDIF
    FOR i = 0 TO r
      READ rank$
    NEXT i
    RESTORE 100
    FOR i = 0 TO s
      READ suit$
    NEXT i
    card$ = rank$ + " of " + suit$
    ccount = ccount + cardvalue
    IF cardvalue = 11 THEN acecount = acecount + 1 \ ENDIF
    WHILE ccount > 21 AND acecount > 0 DO
      acecount = acecount-1
      ccount = ccount-10
    ENDWHILE
    END
    DATA "Ace","Two","Three","Four"
    DATA "Five","Six","Seven","Eight"
    DATA "Nine","Ten","Jack","Queen","King"
100 DATA "Spades","Hearts","Clubs","Diamonds"
```

```
PROCEDURE shuffle
    TYPE d = cards(52),cardnumber:INTEGER
    PARAM deck:d
    DIM i,j,delay,temp:INTEGER

    PRINT "** Dealer Reshuffles **"
    FOR delay = 1 TO INT(RND(10)) + 1
      FOR i = 1 TO 52
        j = RND(51) + 1
        temp = deck.cards(i)
        deck.cards(i) = deck.cards(j)
        deck.cards(j) = temp
      NEXT i
    NEXT delay
    deck.cardnumber = 1
    END
```

PART III

OS-9 commands explained



Welcome to The Official BASIC09 Tour Guide, Part III. Here we'll present the entire set of OS-9 system utilities. You can use them from BASIC09 through the **CHAIN** or **SHELL** statements described in Chapter Three. Each section shows several sample command lines and where practical, a sample run. The commands are listed in alphabetical order to make your life easy.

ATTR

ATTR examines or changes the security attributes of an OS-9 file. It uses the following abbreviations for the attributes.

d	=	**Directory File**
s	=	**non-Sharable file**
r	=	**Read permit to owner**
w	=	**Write permit to owner**
e	=	**Execute permit to owner**
pr	=	**Read permit to public**
pw	=	**Write permit to public**
pe	=	**Execute permit to public**

SAMPLES

```
attr myfile -pr -pw r w <RETURN>
attr sexy_file r w e pr pw pe <RETURN>

attr mydata <RETURN>
-s-wr-wr
```

BACKUP

BACKUP copies all data from one disk to another. It makes a sector by sector copy and pays no attention to file structure. When copying disks, you must make sure that the source and destination disks have the same format. The disks must have the same number of tracks and be initialized to the same density, etc.

BACKUP expects you to give it two device names. If you omit them, it assumes a copy from device /d0 to /d1. If you leave off the second device number, **BACKUP** prompts you for a single drive copy.

BACKUP's options include:

E	**= exit on read error**
S	**= print single drive prompt messages**
-V	**= do not verify**
#nK	**= use thousand bytes of memory**

SAMPLES:

```
backup /d1 /d3 <RETURN>
backup /d0 #20K <RETURN>
backup -v <RETURN>
```

TYPICAL PROMPTING SEQUENCE:

```
OS9: backup <RETURN>

Ready to BACKUP from /d0 to /d1 ?: y
MYDISK is being scratched
OK ?: y
Number of sectors copied: $04FA
Verify pass
Number of sectors verified: $04FA
OS9:
```

BUILD

BUILD lets you enter short text files from the keyboard. The standard input — your terminal — is copied to the path you specify.

BUILD opens a path to the file or device you specify and then prompts you with a question mark, "?".

BUILD exits and returns to OS-9 when you enter a carriage return as the first character in a line.

SAMPLES:

```
build great__procedure <RETURN>
build /p <RETURN>
build <mytext /t2 <RETURN>
```

TYPICAL SEQUENCE:

```
OS9: build a__message <RETURN>

Good Morning <RETURN>
Welcome to OS-9! <RETURN>
<RETURN>

OS9: list a__message <RETURN>

Good Morning
Welcome to OS-9!

OS9:
```

CHD

CHD changes the working data directory. It is an internal **SHELL** command. OS-9 often automatically looks in the working data directory to find text files, programs and other data.

CHX changes the working execution directory. It also is an internal **SHELL** command. OS-9 usually looks in the working execution directory for files that contain code that it can load and run. Because the working execution directory is almost always the "**CMDS**" directory, **CHX** is not used nearly as often as **CHD.**

SAMPLES:

```
chd /d1/sample__programs <RETURN>
chd .. <RETURN>

chx ../cmds <RETURN>
chx /d1/basic <RETURN>
```

TYPICAL SEQUENCE:

```
OS9: chd /d1/BOOK <RETURN>
OS9: dir <RETURN>

     Directory of . 14:57:55
dict          Chapt11      OS9Commands      filefix
Chapt13       Chapt12      Keywords2        SCRATCH03
Printit       Chapt14      Keywords1
```

COBBLER

COBBLER creates a file named "OS9Boot" on a disk loaded in the device named in the command line. The file contains the modules which were loaded into memory during the last boot. **COBBLER** is used when you are making a new customized system disk.

COBBLER may be used only with Level One systems. Level Two systems must use the utility "OS9Gen" to create bootstrap files.

COBBLER expects a contiguous block of disk storage large enough to hold the boot file. You should use freshly formatted disks with it.

SAMPLE:

```
OS9: cobbler /d1 <RETURN>
```

COPY

COPY copies data from the first file or device named to the second.

COPY expects that the first file exists. It creates the second path automatically. **COPY** transfers data in large blocks until it receives and end-of-file signal from the input file.

COPY accepts two modifiers in the command line. The "-s" option lets you perform a single drive copy. With this option, **COPY** prompts you when you should change disks. **COPY** also allows you to use the **SHELL**'s memory size modifier. You'll save time as well as wear and tear on your disk drives when you give copy a lot of memory to use.

SAMPLES:

```
OS9: copy myfile yourfile #15K <RETURN>
OS9: copy /d0/cmds/dir /d1/cmds/dir
```

TYPICAL SINGLE DRIVE SEQUENCE

```
OS9: copy myfile yourfile -s #20K <RETURN>
Ready DESTINATION, hit C to continue: c
Ready SOURCE, hit C to continue: c
Ready DESTINATION, hit C to continue: c
```

DATE

DATE displays the current system date on your terminal. You must have first run the "**SETIME**" command for the time and date to be correct.

DATE also displays the time if you use its "t" option.

SAMPLES:

```
OS9: date <RETURN>
OS9: date t>/p <RETURN>
```

TYPICAL SEQUENCE:

```
OS9: date t

March 6, 1983     15:25:40
```

DCHECK

DCHECK is used to check a diskette for flaws. It checks the general integrity of the linkage between your directories and files. **DCHECK** also spots sectors on a disk that have been marked as allocated but in fact are not associated with a file.

DCHECK expects the name of a disk device as a parameter.

DCHECK allows six options in the command line:

-w=<path>	**pathlist to directory for work files**
-p	**print pathlists for questionable clusters**
-m	**save allocation map work files**
-b	**suppress listing of unused clusters**
-s	**display count of files and directories only**
-o	**print DCHECK's valid options**

SAMPLE:

```
OS9: dcheck /d1 <RETURN>
```

TYPICAL SEQUENCE:

```
OS9: dcheck /d1 <RETURN>

Volume—"The.Book" on device /d1
$009A bytes in allocation map
1 sector per cluster
$0004FA total sectors on media
Sector $000002 is start of root directory FD
$0010 Sectors used for id, allocation map and root directory
Building allocation map file...
Checking allocation map file...

"The.Book" file structure is intact
1 directory
11 files
```

DEL

DEL deletes file(s) named in the command line. You must have write permission for a file before you may delete it.

DEL allows one option, -x. This option tells OS-9 to look for the file in the current execution directory.

DEL may not be used to delete directory files unless their type is changed to non-directory. It's better to use **DELDIR** for this.

SAMPLES:

```
OS9: del myprogram hisprogram <RETURN>
OS9: del /d1/BOOK/Chapt2 <RETURN>
```

TYPICAL SEQUENCE:

```
OS9: dir /d1 <RETURN>

Directory of /d1 15:46:57
myfile          hisfile

OS9: del /d1/myfile <RETURN>
OS9: dir /d1 <RETURN>

   Directory of /d1          15:47:55
   hisfile
```

DIR

DIR displays a formatted listing of the files in a directory on your terminal.

DIR allows two options on the command line. The "x" option lists the current execution directory. The "e" option gives you a complete description of all files. It shows you the size, address, owner, permissions, and the date and time the file was last modified.

SAMPLES:

```
OS9: dir <RETURN>
OS9: dir x <RETURN>
OS9: dir x e <RETURN>
OS9: dir /d1/basic_programs <RETURN>
OS9: dir /d1/basic_listings e <RETURN>
```

DISPLAY

DISPLAY is used to send special characters such as cursor or screen control codes to your terminal or printer. It lets you send characters that cannot be **ECHO**ed.

DISPLAY converts the hexadecimal number you enter to **ASCII** and echos it to the standard output device.

DISPLAY accepts one or more hexadecimal numbers.

SAMPLES:

```
OS9: display 0C 1F 02 7F <RETURN>
OS9: display 15 >/p <RETURN>
```

TYPICAL SEQUENCE:

```
OS9: display 41 42 43 44 45 46 47 <RETURN>
ABCDEFG
```

DSAVE

DSAVE creates a procedure file that lets you copy all files from a directory or device at once.

DSAVE automatically writes copy commands to copy files from the current data directory to the directory specified in the command line. These commands go to the standard output device. If **DSAVE** encounters a directory, it automatically generates a **MAKDIR command and changes the working data directory.**

DSAVE accepts six options in the command line:

-b	**make output disk a system disk and use source disk's "OS9Boot" file.**
-b = <path>	**same as "-b" except use "path" as source for "OS9Boot" file.**
-i	**indent for directory level**
-l	**do not process directories below the current level**
-m	**do not include MAKDIR commands in procedure file**
-s <integer>	**set copy size parameter to <integer> K**

TYPICAL SEQUENCE:

```
OS9:  chd /d1 <RETURN>
OS9:  dsave /d1 >/d0/makecopy <RETURN>
OS9:  chd /d1 <RETURN>
OS9:  /d0/makecopy <RETURN>
```

DUMP

DUMP sends a formatted display of the data from a path to the standard output device. It gives you a way to examine the contents of files that do not contain text.

DUMP displays 16 bytes of data on each line. Both the hexadecimal and ASCII value are printed. Data bytes that are non-printable in ASCII are represented by periods.

DUMP displays addresses relative to the beginning of a file. Since memory modules are stored in files in exactly the same form that they exist in memory, the addresses displayed by **DUMP** correspond to relative load addresses within a memory module.

SAMPLES:

```
OS9:  dump <RETURN>      (displays keyboard)
OS9:  dump myfile >/p <RETURN>
OS9:  dump ourfile <RETURN>
```

```
Addr   0 1   2 3   4 5   6 7   8 9   A B   C D   E F   0 2 4 6 8 A C E
0000   5468  6973  2069  7320  6120  736D  616C  6C20  This is a small
0010   6578  616D  706C  6520  7465  7874  2066  696C  example text file
0020   6520  7768  6963  6820  6973  2070  7269  6E74  which is printed
0030   6564  2069  6E20  6269  6E61  7279  2061  6E64  in binary and
0040   2041  5343  4949  2062  7920  7468  6520  2264  ASCII by the
0050   756D  7022  2063  6F6D  6D61  6E64  2E0D        "dump" command..
```

ECHO

ECHO sends messages to the standard output path. It is often used in procedure files to send a message to the operator.

ECHO should not be used to echo any of the standard punctuation characters used by the **SHELL** such as <, >, !, &, etc.

SAMPLES:

```
OS9: echo >/t1 Let's go to lunch John <RETURN>
OS9: echo >/term ** WARNING ** Scratching disk
```

TYPICAL SEQUENCE:

```
OS9: echo System will be shut down in 10 minutes
System will be shut down in 10 minutes
```

EX

EX lets you execute another program from the **SHELL** without starting a new process to save system memory.

Basically, the Shell runs the command **EX** and then zaps itself. For this reason, **EX** must always be the last command on a **SHELL** input line because any commands following it will never be processed.

SAMPLES:

```
OS9: ex BASIC09 <RETURN>
```

FORMAT

FORMAT initializes, verifies, and sets up the initial file structure on a new or recycled disk. **FORMAT** must be run on all new disks before they can be used by an OS-9 system.

FORMAT reads a set of default values for the disk from the device descriptor. However, you may **FORMAT** a disk differently by supplying options in the comand line. Even though many formats are possible, you may be limited by the capabilities of your disk controller and drives. Consult Microware's OS-9 Operating System Users Manual to learn the details of **FORMAT**'s operation.

FORMAT's command line options for floppy disks are:

S	**= single density**
D	**= double density**
1	**= single sided**
2	**= double sided**
R	**= ready**

Other options include:

'number'	**= number of tracks in decimal**
:number:	**= number of sector interleave value**
"name"	**= disk name (32 characters maximum)**

SAMPLES:

```
OS9: format /d1 2 D "database" '77' <RETURN>
OS9: format /d1 S 1 <RETURN>
```

TYPICAL SEQUENCE:

```
OS9: format /d1

                     FORMAT 1.1

  TABLE OF FORMAT VARIABLES

              Recording Format:   MFM <- density: FM=sgl MFM=dbl
             Track density in TPI:  48 <- some 5" disks use 96 TPI
            Number of Cylinders:  77 <- 77 for 8", 35/40/80 for 5"
            Number of Surfaces:   1 <- 1 or 2 sides
         Sector Interleave Offset:   3 <- set by manufacturer

                     Disk type:   8 <- 5, 8, or HARD
  Sectors/Track on TRK 0, Side 0:  16 <- set by manufacturer
                 Sectors/Track:  28 <- set by manufacturer

Formatting on drive /d1
Y (yes), n (no), or q (quit) y          <- answer: y to format, n to
Ready: Y                                   change table, or q to stop
Disk Name: Documentation                <- enter up to 32 characters

(track numbers are printed here)

 GOOD SECTOR COUNT = $860
```

FREE

FREE reports the number of sectors of free space remaining on a disk. **FREE** also tells you the name of the disk, the date the disk was created and the cluster size.

FREE reports the cluster size so that you can determine how many new files will fit on a disk. For example, some disk drives use 8 sectors per cluster. On these drives — if **FREE** reports 32 free sectors — you only have room for four new files.

SAMPLE RUN:

```
OS9: free /d1 <RETURN>
Documentation created on: 82/08/26
Capacity: 1,274 sectors (1-sector clusters)
1,070 free sectors, largest block 940 sectors
```

IDENT

IDENT displays the header information from an OS-9 memory module. It prints the module size and its **CRC.** If the module is a program or device driver module, **IDENT** reports the execution offset and the permanent data storage requirement.

IDENT prints the type/language and attribute/revision bytes on one line and interprets them on the next. If **IDENT** is run on a disk file, it reports on each module in the file.

IDENT has three command line options:

-v = do not verify module CRC
-s = use short form
-m = assume that path is a module in memory

SAMPLE RUN:

```
OS9: ident -m ident
Header for:         Ident
Module size:        $06A5         #1701
Module CRC:         $1CE78A      (Good)
Hdr parity:         $8B
Exec. off:          $0222          #546
Data Size:          $0CA1         #3233
Edition:            $05              #5
Ty/La At/Rv:        $11   $81
Prog mod, 6809 obj, re-en
```

KILL

KILL sends an abort signal to the process having the process ID number named in the command line. You cannot **KILL** a process unless you have the same user ID as the user that started the process.

SAMPLES:

```
OS9: kill 5 <RETURN>
OS9: kill 12 <RETURN>
```

LINK

LINK locks a previously loaded module into memory. On Level Two systems it also maps the named module into your address space.

LINK increments the link count of a module each time it is run.

SAMPLES:

```
OS9: link edit <RETURN>
OS9: link basic09 <RETURN>
```

LIST

LIST copies text lines from all input files (paths) given in the command line to the standard output path. It is used to examine or print text files.

LIST runs until it receives an end-of-file signal from the last input path.

SAMPLES:

```
OS9: list /d0/startup >/p <RETURN>
OS9: list /d1/BOOK/chapt14 /d1/BOOK/Chapt15 <RETURN>
OS9: list /term >/p
```

LOAD

LOAD opens a file and loads all its modules into memory. It then adds the name of the modules loaded into OS9's module directory. If a module is loaded that is already in memory, **LOAD** will keep the module with the highest revision number.

SAMPLE:

```
OS9: load Basic09 <RETURN>
```

LOGIN

LOGIN provides security for timesharing systems. **TSMON,** OS-9's timesharing monitor, calls it automatically.

LOGIN requests a user name, and a password and checks them against a validation file. You have three chances to answer each question correctly before the process is aborted.

LOGIN automatically sets up your user number, working execution directory, working data directory, and executes a program specified in the password file.

LOGIN's validation file is complex. See your OS-9 Operating System Users Manual for complete details.

SAMPLE RUN:

```
OS9: login <RETURN>

OS-9 Level 1 Timesharing System Version 1.2 83/03/07 21:25:28
User Name? michele <RETURN>
Password? tiffy <RETURN>

Process #04 logged     83/03/07 21:26:05
Welcome!
```

MAKDIR

MAKDIR creates a new directory file on a disk. The name in the pathlist is used as the name of the directory.

MAKDIR will not create a directory for you unless you have write permission for its parent directory. Many programmers like to capitalize the names of directories to make them stand out from file names.

SAMPLES:

```
OS9: makdir /d1/BOOK <RETURN>
OS9: makdir HOMEWORK <RETURN>
OS9: makdir ../MUSIC_LIBRARY <RETURN>
```

MDIR

MDIR displays the names of modules residing in memory.

MDIR has one command line option, "e", which lets you list the physical address, size, type, revision level, and user count of each module. **MDIR** prints this information in hexadecimal form.

MDIR also displays the extended physical address of a module when used on a Level II system.

SAMPLE RUN:

```
OS9: mdir <RETURN>

          Module Directory at 21:42:16
OS9p2     Init      Boot         OS9          Ioman
RBF       SCF       Sysgo        ACIA         PIA
TERM      T1        P1           P            Clock
Shell     G68
```

MERGE

MERGE copies the multiple input files named in your command line to the standard output path. Merge lets you combine several input files into one output file by redirecting the standard output path.

MERGE does no output line editing. For example, it does not automatically insert line feeds after carriage returns, etc.

SAMPLES:

```
OS9: merge file1 file2 file3 >allfiles <RETURN>
OS9: merge original_file new_file >/p
```

MFREE

MFREE displays a list of memory areas available for use. It reports both the address and size of each free memory block. On a Level I system **MFREE** reports the number of 256-byte pages available.

MFREE also shows the block number, physical beginning and ending addresses, and size of each memory area when run on Level II systems. The Level II size is reported as both the number of blocks and the number of free (K)ilobytes available.

SAMPLE RUN:

```
OS9: mfree <RETURN>

Address pages
 800-  8FF    1
 B00-AEFF  164
B100-B1FF    1

Total pages free = 166
```

OS9GEN

OS9Gen creates and links the "OS9Boot" file which must be on any disk used to "boot" the system. You may use it to simply make a copy of an existing boot file, to add modules to an existing boot file, or to create an entirely new boot file.

OS9Gen receives the name of the device where the OS9Boot file is being installed from the command line. It copies a list of files to a file on that device, names it "OS9Boot", and links to it.

OS9Gen's operation is quite complex and well beyond the scope of a beginner. If you're interested in how it works, consult the detailed description given in the OS-9 Users Manual.

SAMPLES:

```
OS9: os9gen /d1 <RETURN>                       ;* run os9gen
OS9: /d0/os9boot <RETURN>                      ;* file we're installing
OS9: <ESCAPE>                                  ;* end of file

OS9: os9gen /d1 <RETURN>
OS9: /d0/os9boot <RETURN>                      ;* first file
OS9: /d1/new_video_drivers <RETURN>            ;* second file
OS9: /d1/new_modem_drivers <RETURN>            ;* yet another file
OS9: <ESCAPE>
```

PRINTERR

PRINTERR displays English language error messages from the file /d0/SYS/errmsg. It replaces the standard OS-9 error reporting routine which only prints error code numbers.

PRINTERR installs itself permanently the first time you run it and it may not be undone. The OS-9 Users Manual shows you how to change the existing error message file or install your own.

SAMPLE:

```
OS9: printerr <RETURN>
```

PROCS

PROCS displays the list of processes currently running on your system.

PROCS only lists the processes you own unless you ask to see all by using the "e" option in the command line

PROCS shows you the user and process ID numbers, state, priority, memory size in 256-byte pages, the primary program module, and the standard input path for the process when running on Level I systems.

PROCS gives the process ID number, ID number of the parent process, user index, process priority, memory size in 256-byte pages, current stack pointer address, and the primary module name on Level II systems.

SAMPLES:

```
OS9: procs <RETURN>
OS9: procs e <RETURN>
```

RENAME

RENAME changes the name of a file. You must have write permission for a file before you can rename it.

RENAME may not be used to change the name of a device or the name of the current data or parent directories, "." or ".."

SAMPLE:

```
OS9: rename outstanding for_sure <RETURN>
OS9: rename /d1/TEXT/speech soapbox <RETURN>
```

SAVE

SAVE creates a new file and writes a copy of the memory module(s) named in the command line. These modules must exist in the module directory when you run **SAVE**. This command is often used to save a copy of a program module after it has been "patched".

SAVE uses the current data directory as its default directory.

SAMPLES:

```
OS9: save/d0/CMDS/filefix filefix <RETURN>
OS9: save /d0/CMDS/math_routines add sub mult div <RETURN>
```

SETIME

SETIME sets the date and time and activates the system's real time clock. You may enter input to **SETIME** from the command line or you may answer a prompt.

SETIME must be run before you can do any multitasking. If you have a battery backed up clock in your system, you need only run **SETIME** with the year as the parameter. It will get the rest of its information from the clock.

SAMPLES:

```
OS9: setime 83,03,08,2143 <RETURN>
OS9: setime 830308 214535 <RETURN>
OS9: setime 83 <RETURN>
```

SLEEP

SLEEP puts a process to sleep for the requested number of ticks. You may use it to generate time delays or to "break up jobs that are CPU intensive. The length of a tick depends upon which system you are running. The typical tick on a Level One system lasts 1/10 second. With Level Two most systems use a 1/100 second tick.

SAMPLE:

```
OS9: sleep 10 <RETURN>
```

SETPR

SETPR is a built in **SHELL** command that changes the CPU priority of a process. You may only use it with a process that has your user ID number.

SETPR's priorities range from 1 (the lowest) to 255.

SAMPLE:

```
OS9:setpr 4 200 <RETURN>
```

SHELL

SHELL is the command interpreter that reads data from the standard input path — usually your keyboard — and interprets it as a sequence of commands.

SHELL's technical specifications and syntax are explained in great detail in Microware's OS-9 Users Manual.

TEE

TEE copies all text lines from the standard input path to the standard output path and to any other paths named in the command line.

TEE is a filter that may be used with a pipeline to simultaneously send a listing to your terminal, printer and a disk file — or any number of destinations.

SAMPLES:

```
OS9: dir ! tee /p /d1/WORKTEXT/scratch <RETURN>
OS9: echo Let's go to lunch ! tee /t1 /t2 /t3 <RETURN>
```

TMODE displays or changes the operating parameters of your terminal. If you do not give **TMODE** arguments in the command line, it lists the current state of each parameter.

TMODE processes parameters named in the command line. It works on the standard input path unless an optional path is named with a period, ".", and a number in the command line.

TMODE's parameters include:

PARAMETER	**MEANING TO TERMINAL:**
upc	**Upper Case Only**
-upc	**Upper and Lower Case**
bsb	**Erase on Backspace**
-bsb	**Does Not Erase on Backspace**
bsl	**Backspace over line**
-bsl	**No Backspace Over Line (use linefeed)**
echo	**Echo Input characters to terminal**
-echo	**Do Not Echo Characters**
lf	**Issue line feed with each return**
-lf	**Do Not send line feed with return**
pause	**Stop when screen full**
-pause	**Do not stop when screen full**
null=n	**Issue "n" nulls after return**
pag=n	**Set video page length to "n" lines**
bsp=h	**Define backspace character as "h"**
bse=h	**Define backspace echo character as "h"**
del=h	**Define character that deletes a line**
bell=h	**Define bell character**
eof=h	**Define end of file input character**
eor=h	**Define end-of-record input character**
type=h	**Initialize ACIA with "h"**
reprint=h	**Define reprint line character.**
dup=h	**Define last input line character**
psc=h	**Define pause character**
abort=h	**Define abort character**
quit=h	**Define quit character**
xon=h	**Define character used to resume transmission**
xoff=h	**Define character to suspend transmission**

n = decimal number
h = hexadecimal number

SAMPLES:

```
OS9: tmode -upc -lf null=0 pause <RETURN>
OS9: tmode pag=12 pause bsl bsp=8 <RETURN>
```

TSMON

TSMON monitors terminals on timesharing systems. It supervises idle terminals and initiates a login sequence when a carriage return is typed. You may log off the system by sending an end-of-file character — usually <**ESCAPE**> — as the first character of a command line.

SAMPLES:

```
OS9: tsmon /t18 <RETURN>
```

UNLINK

UNLINK tells OS-9 that you are through with a module. If you are the only user, **UNLINK** removes the module from the module directory and gives the memory back to the system. If no other operators are using the module, **UNLINK** simply lowers the module's link count by one.

WARNING: Never **UNLINK** a module that you did not **LOAD** or **LINK** to.

SAMPLES:

```
OS9: unlink program1 program5 program9 <RETURN>
```

VERIFY

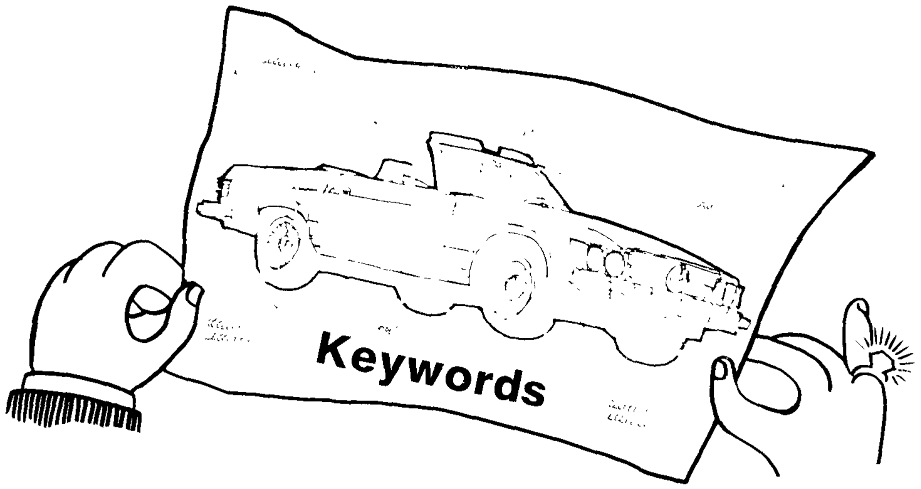
VERIFY checks the module header parity and CRC of all modules in a file. The modules are read from the standard input path. Any error messages are sent to Standard error path.

VERIFY's update, "U", option causes OS9 to copy a module to the standard output path and compute new values for the header parity and CRC. **VERIFY** does not copy the module to standard output unless you specify this option.

SAMPLES:

```
OS9: verify <myprogram <RETURN>
Module's header parity is correct.
Module's CRC is correct.

OS9: verify <Oldfile >Newfile u <RETURN>
```

BASIC09 keywords explained



Welcome to The Official BASIC09 Tour Guide, Part IV. Here you'll find a precise definition of each BASIC09 keyword. We hope it becomes one of the most useful reference works in your library.

We organized each keyword entry in the same manner to help you find your answers fast. Each entry classifies the keyword according to its function and presents its formal syntax.

The first paragraph following the syntax tells you what to expect when you execute the keyword, the second tells you what information you must provide, and the third explains a typical use of the keyword.

Where possible, a sample procedure containing the keyword follows the narrative description. And, in cases where there is a visible result, we show you a sample run of the procedure. Finally, we present a list of other BASIC09 keywords related to the word in the specific entry.

MATH FUNCTION

SYNTAX: ABS(<NUM>)

ABS returns the absolute value of the number or variable in parentheses. A number's absolute value is its value without a + or— sign.

ABS needs one argument, a number or variable of type **REAL** or **INTEGER.**

ABS is used in arithmetic operations when a non-negative result is needed.

EXAMPLE:

```
PROCEDURE absdem
(* Sample procedure for ABS *)
DIM number,another_number:REAL

(* First, we'll print some ABSolute numbers *)
PRINT
PRINT ABS(-476.35),ABS(-.0031459),ABS(798.56)

(* Then, we'll use ABS in a mathematical expression *)
number: = 37
another_number: = 59

PRINT
PRINT "The ABSolute value of "; (number-another_number)/2; " is ";
ABS((number-another_number)/2)
PRINT

END
```

See also: **SGN, SQ, VAL**

MATH FUNCTION

SYNTAX: ACS(<NUM>)

ACS returns the arcosine of a number of type **REAL** or **INTEGER** in degrees or radians.

ACS needs one argument which is a number or variable of type **INTEGER** or **REAL.**

ACS is used in mathematic formulas to find the inside angles of a right triangle. The angle is determined by the ratio between the adjacent side and the hypotenuse.

EXAMPLE:

```
PROCEDURE acsdem
(* Demonstrate use of ACS trig function *)
(* The angle created by a given ratio between *)
(* the side of a triangle next to the angle *)
(* and the hypotenuse *)

(* Use Degrees *)

DEG

DIM nearside,hypotenuse,ratio:REAL

INPUT "The length of the side adjacent to the angle? ",nearside
INPUT "The length of the hypotenuse? ",hypotenuse

ratio:=nearside/hypotenuse

PRINT "The angle between an adjacent side of "; nearside
PRINT "and a hypotenuse of "; hypotenuse; " is ";
PRINT ACS(ratio); " degrees."
PRINT

END
```

See also: **SIN, COS, TAN, ASN, ACS, ATN**

ADDR

MISCELLANEOUS FUNCTION

SYNTAX: ADDR(<NAME>)

ADDR returns an **INTEGER** value which is the absolute memory address of the **VARIABLE, ARRAY,** or **STRUCTURE** named in the parentheses.

ADDR needs only one parameter — the **NAME** of a **VARIABLE, ARRAY,** or **STRUCTURE.**

EXAMPLE:

```
PROCEDURE addrdem
(* Demonstrate use of ADDR *)
DIM address,count:INTEGER
DIM sampler:STRING
DIM characterholder:BYTE

(* Assign value to string *)
sampler:="The Quick Brown Fox"

(* Now point directly to STRING in memory *)
address:=ADDR(sampler)

(* Get each character from memory and PRINT it *)
FOR count:=0 TO LEN(sampler)
characterholder:=PEEK(address+count)
PRINT CHR$(characterholder);
NEXT count
PRINT

END
```

See also: **none**

AND

BOOLEAN OPERATOR

SYNTAX: <EXPRESSION> AND <EXPRESSION>

AND returns a **BOOLEAN** value of **TRUE** or **FALSE.**

AND needs two arguments written in an expression.

AND is used as a logical math operator in a **BOOLEAN** expression. It has a higher precedence than **OR,** but less precedence than **NOT.**

EXAMPLE:

```
PROCEDURE andemo
(* demonstrate use of boolean AND *)
DIM onenumb,twonumb,threenumb:INTEGER

onenumb:=1
twonumb:=2
threenumb:=3

IF onenumb<twonumb AND threenumb>twonumb THEN
PRINT "That's right, 1<2 and 3>2!"
ELSE
PRINT "Math always was hard for me..."
ENDIF
END
```

See also: **OR, NOT, BOOLEAN, FALSE, TRUE**

STRING FUNCTION

SYNTAX: ASC(<STRING>)

ASC returns the ASCII value of the first character of the **STRING** in parentheses. This value will always be between 1 and 255.

ASC needs only one parameter — a **STRING** or **STRING** variable.

ASC converts a **STRING** or **STRING** variable to its corresponding ASCII decimal number.

EXAMPLE:

```
PROCEDURE ascdem
(* Demonstrate ASC function *)
(* Convert String or Character to ASCII decimal number *)

DIM characterholder:STRING[1]
DIM count:INTEGER

PRINT
INPUT "Type any character, then <RETURN>: ",characterholder
PRINT "The ASCII code for the letter "; characterholder;
PRINT " is "; ASC(characterholder)
END
```

See also: **LEN, SUBSTR**

MATH FUNCTION

SYNTAX: ASN(<NUM>)

ASN returns the arcsine of the number in parentheses. The result is always a **REAL** number and is expressed in degrees or radians — depending on the value of the function flag set by **DEG** or **RAD.**

ASN needs one parameter which may be a **REAL** or **INTEGER** number.

ASN is used to determine the angles inside a right triangle.

EXAMPLE:

```
PROCEDURE asndem
(* Demonstrate ASN function *)
(* The angle created by a certain ratio between *)
(* the side of a triangle opposite the angle and *)
(* the hypotenuse *)

(* Use DEGrees *)

DEG

DIM oppositeside,hypotenuse,ratio:REAL

INPUT "Length of the side opposite the angle? ",oppositeside
INPUT "How long is the hypotenuse? ",hypotenuse

ratio:=oppositeside/hypotenuse

PRINT "The angle formed in a triangle with an opposite side of "
PRINT oppositeside; " and a hypotenuse of "; hypotenuse
PRINT "is "; ASN(ratio); " degrees."
END
```

See also: **SIN, COS, TAN, ACS, ATN**

MATH FUNCTION

SYNTAX: ATN(<NUM>)

ATN returns the arctangent of the number in parentheses. The result is a **REAL** number and it may be expressed in degrees or radians — depending on the value of the function flag set by **DEG** or **RAD.**

ATN needs one parameter. It may be a **REAL** or **INTEGER** number or a **VARIABLE.**

ATN is used to calculate an angle inside a right triangle. It is the opposite of the math function of **TAN.**

EXAMPLE:

```
PROCEDURE atndem
(* Demonstrate ATN function *)
(* Determine the angle in a triangle *)
(* given the two sides *)

DIM opposite,adjacent,ratio,angle:REAL
(* Switch BASIC09 to degrees instead of radians *)

DEG

INPUT "Length of opposite side? ",opposite
INPUT "Length of adjacent side? ",adjacent

ratio:=opposite/adjacent
angle:=ATN(ratio)

PRINT
PRINT "Your angle is "; angle; " degrees."
END
```

See also: **TAN, SIN, COS, ASN, ACS**

DIRECTIVE STATEMENT

SYNTAX: BASE 0
BASE 1

BASE determines if the lowest array or data structure index (subscript) is zero or one.

BASE needs only one argument — a "1" or a "0".

BASE defaults to one and does not affect string operations. The beginning character of a **STRING** always has an index of one.

EXAMPLE:

```
PROCEDURE basedem
(* Demonstrate use of BASE statement *)
(* BASE sets the lowest variable array *)
(* element to zero or one *)

DIM index,anarray(6):INTEGER


(* First let's show operation in BASE zero *)

BASE 0

(* First initialize the array *)

FOR index:=0 TO 5
anarray(index):=index
NEXT index


(* Now print the array *)

PRINT
FOR index:=0 TO 5
PRINT anarray(index);
PRINT " ";
NEXT index
PRINT


(* Now print same array using BASE one *)
```

```
BASE 1

FOR index:=1 TO 6
PRINT anarray(index);
PRINT " ";
NEXT index

PRINT
PRINT
PRINT "Notice that the value of array elements "
PRINT "does not change when we change the name of "
PRINT "the elements."
```

See also: **none**

BOOLEAN

TYPE DECLARATION

SYNTAX: DIM <VARIABLE>: BOOLEAN

BOOLEAN is used in **DIM** and **TYPE** statements to declare variables of type **BOOLEAN**.

BOOLEAN variables return one of two values, either **TRUE** or **FALSE**. They may not be used for numeric computation.

See also: **AND, OR, XOR, NOT, TRUE, FALSE**

BREAK

DEBUG COMMAND

SYNTAX: BREAK <PROCEDURE NAME<

BREAK inserts a breakpoint in the procedure named.

BREAK needs only one argument — the name of a procedure.

BREAK is used with BASIC09's **DEBUG** mode. It is entered as a command from the keyboard.

See also: **STATE**

BYE

CONTROL STATEMENT

SYNTAX: BYE

BYE stops the procedure being run and exits BASIC09. It closes any open files. It can be dangerous if you have not saved your program before it is used.

BYE needs no arguments.

BYE is used to return to OS-9.

EXAMPLE:

```
PROCEDURE byedem
(* Show use of BYE to return to OS-9 *)
(* operating system from a program *)

PRINT "Let's return to OS-9 now."
PRINT "Type 'BASIC09' to return."
PRINT "Goodbye!"
BYE
```

See also: **PAUSE, END, STOP**

BYTE

TYPE DECLARATION

SYNTAX: DIM <VARIABLE>: : BYTE

BYTE is used in **DIM** and **TYPE** statements to declare variables of type **BYTE**.

BYTE variables return unsigned whole numbers ranging from zero to 255. They are stored in one memory location. **BYTE** variables need only half the storage used by integers and one-fifth that used by reals.

See also: **INTEGER, REAL, BOOLEAN**

OS9 SYSTEM CALL

SYNTAX: CHAIN <STRING>

CHAIN exits Basic09 to run an OS-9 program (which can be another Basic09 program).

CHAIN needs only one argument — a string expression that holds the name of the OS-9 program and any arguments it requires.

CHAIN is often used to run selected programs from a menu-driven program.

EXAMPLE:

```
PROCEDURE chaindem

(* Show use of CHAIN statement *)
(* It is best to use the Shell's 'ex' option *)
(* when you are CHAINing to an OS-9 program. *)

PRINT "Let's see what is in our current data directory."
PRINT "Notice that when you return from OS-9, your BASIC09"
PRINT "workspace will be empty."
PRINT
PRINT "This happens because you leave BASIC09 completely, use"
PRINT "OS-9's Shell to run the DIR utility and then return"
PRINT "to BASIC09 again."

CHAIN "dir ; ex basic09"
```

See also: **none**

DIRECTIVE STATEMENT, SYSTEM COMMAND

SYNTAX: CHD STRING

CHD changes the current **DATA** directory.

CHD needs one argument — a **STRING** expression which specifies the pathlist of a directory file.

CHD is explained in detail in Microware's OS-9 User's Guide.

See also: **CHD**

STRING FUNCTION

SYNTAX: CHR$(<INT>)

CHR$ returns the ASCII character represented by an **INTEGER** value. In other words, it converts the ASCII code for a character to the character itself.

CHR$ needs one parameter — an **INTEGER** constant or variable with a value between 1 and 255.

CHR$ is used with the **PRINT** statement to send special non-printable control codes to your terminal's screen or a printer.

EXAMPLE:

```
PROCEDURE chrdemo
(* demonstrate the use of CHR$ *)
DIM a,b,c,d,e,f,g,h,i:BYTE
DIM count:INTEGER

(* assign value to each variable *)
a:=67
b:=97
c:=116
d:=61
e:=77
f:=69
g:=79
h:=87
i:=33

(* Then PRINT its character value *)
PRINT CHR$(a); CHR$(b); CHR$(c); CHR$(d); CHR$(e); CHR$(f); CHR$(g); CHR$(h)
CHR$(i)

(* Or use a loop to set character value *)
FOR count:=65 TO 90
PRINT CHR$(count);
NEXT count
PRINT

END
```

See also: **LEFT$, RIGHT$, MID$, TRIM$, DATE$**

**DIRECTIVE STATEMENT,
SYSTEM COMMAND**

SYNTAX: CHX<STRING>

CHX changes the current **EXECUTION** directory.

CHX needs one argument — a **STRING** expression which specifies the pathlist of the directory file.

CHX is explained in detail in Microware's OS-9 User's Guide.

EXAMPLE:

```
PROCEDURE chxdem
(* Show use of CHX statement to change execution *)
(* directory from within a program *)

SHELL "load pxd"
PRINT "Your current execution directory is: ";
SHELL "pxd"

PRINT
PRINT "Now, let's change it."

CHX "/d0"

PRINT "And, confirm that it has been changed."
PRINT
PRINT "Your current execution directory is now: ";

SHELL "pxd"

PRINT
PRINT "Don't forget to change it back."
```

See also: **CHD**

CLOSE

INPUT/OUTPUT STATEMENT

SYNTAX: CLOSE <INTEGER EXP> , <INTEGER EXP>

CLOSE shuts down the path defined by the **INTEGER** expression which was previously **OPEN**ed or **CREATE**d when it is no longer needed.

CLOSE may have several arguments, but each path number passed must be an integer expression.

CLOSE should always be used when you finish reading or writing a file. **CLOSE** may also be used to release non-sharable devices — a printer for example — to other users.

EXAMPLE:

```
PROCEDURE closedem
(* Show use of Close statement *)
(* We'll open a path to the system printer, *)
(* send a short sentence and then Close the path. *)

DIM path:INTEGER

OPEN #path,"/p"
PRINT #path,"Hello There"
CLOSE #path
```

See also: **CREATE, OPEN**

CONT

DEBUG MODE COMMAND

SYNTAX: CONT

CONT is used from BASIC09's Debug Mode to continue execution of a procedure.

CONT needs no arguments and is entered from the terminal.

CONT is sometimes used in conjunction with **BREAK** to debug a program.

EXAMPLE:

```
PROCEDURE contdem
(* Show use of CONT statement from a program *)
```

```
DIM count:INTEGER

FOR count:=1 TO 10 \ PRINT count; \NEXT count

PRINT
PRINT
PRINT "Type CONT to continue this program."

PAUSE

PRINT
PRINT "The program has passed the STOP statement and"
PRINT "continues to run."

PRINT \ PRINT

FOR count:=1 TO 10 \ PRINT count; \NEXT count

PRINT \ PRINT \ PRINT "Goodbye!"
```

See also: **BREAK**

COS

MATH FUNCTION

SYNTAX: COS(<NUM>)

COS returns the cosine of the specified angle as a **REAL** number. The **NUM** may be in degrees or radians — depending on the flag set by **DEG** or **RAD**.

COS needs one parameter — a **REAL** or **INTEGER** number.

COS is used to find the length of one side of a triangle if the other two sides and an angle are known.

EXAMPLE:

```
PROCEDURE cosdem
(* Demonstrate the COS function *)

DIM angle:INTEGER

DEG

INPUT "Enter an angle expressed in degrees: ",angle
PRINT "The cosine of a "; angle; " degree angle is ";
PRINT COS(angle)
PRINT
```

See also: **ACS, ATN, SIN, TAN, ASN**

INPUT/OUTPUT STATEMENT

SYNTAX: CREATE <INT VAR>,<STRING EXP> [: <ACCESS MODE>]
<ACCESS MODE> := <MODE> ! <MODE> +
<ACCESS MODE> <MODE>:= WRITE ! UPDATE ! EXEC

CREATE is used to create a new file. When files are created, they may be set up in the **WRITE, UPDATE** or **EXEC** modes.

CREATE needs several arguments — an **INTEGER** variable, a **STRING** expression, and an optional access mode.

CREATE is explained in detail in Chapter 10.

EXAMPLE:

```
PROCEDURE createdem
(* Show CREATE statement in action *)
(* We'll Create a file, write something *)
(* to it, read it and finally, delete it *)

DIM file:INTEGER

PRINT "Let's CREATE a file and put a sentence in it."
PRINT

CREATE #file,"demofile":UPDATE
PRINT #file,"This message is going into a demo file."
CLOSE #file

PRINT "Now, we'll use OS-9's DIR utility to see if it's there"
PRINT "and the LIST utility to view it."

SHELL "dir"
PRINT
SHELL "list demofile"

PRINT \ PRINT "We're back in BASIC09, so let's delete"
PRINT "the file."

SHELL: "del demofile"
```

See also: **WRITE, UPDATE, EXEC, GET, PUT, PRINT**

INPUT/OUTPUT STATEMENT

SYNTAX: DATA <expression>,{<expression>}

DATA statements are used to build constant tables within a program to be accessed later by **READ** statements.

DATA uses a list of one or more expressions separated by commas, which can be of any type.

Basic09 is unusual compared to other Basics in that it allows **DATA** statements to include expressions and variables.

EXAMPLE:

```
PROCEDURE datadem
(* Show use of DATA statement *)

DIM word:STRING[16]
DIM count,number:INTEGER

DATA 1," You ",2," are ",3," printing ",4," numeric "
DATA 5," and ",6," string ",7," data."

PRINT

FOR count:=1 TO 7
READ number
PRINT number;
READ word
PRINT word
NEXT count

PRINT
```

See also: **READ, RESTORE**

DATE$

STRING FUNCTION

SYNTAX: DATE$ "yy/mm/dd <hh:mm:ss>"

DATE$ returns the year, month, and day, including hours, minutes, and seconds from your system's clock.

DATE$ returns a string value which contains the year, month, day, hours, minutes, and seconds in fixed format.

EXAMPLE:

```
PROCEDURE datedem
(* print the date *)
DIM dateholder:STRING[17]
dateholder: = DATE$

(* Print the date and time direct *)
PRINT "Ta da! Today's date is ... "; DATE$

(* And show that it has also been stored in dateholder *)
PRINT \ PRINT dateholder \ PRINT

END
```

See also: **CHR$, LEFT$, RIGHT$, MID$, TRIM$**

DEG

**DIRECTIVE STATEMENT,
SYSTEM COMMAND**

SYNTAX: DEG

DEG tells a procedure that angles are being expressed in degrees when executing **SIN, COS, TAN, ACS, ASN,** or **ATN** functions.

DEG needs no arguments. It may be used in a procedure or typed from the keyboard while in the **DEBUG** mode.

DEG is used to toggle a procedure's state from **RAD** (radians).

EXAMPLE:

```
PROCEDURE degdem
(* Demonstrate use of DEG statement *)
(* Use Degrees *)
DEG
```

```
DIM nearside,hypotenuse,ratio:REAL

INPUT "The length of the side adjacent to the angle? ",nearside
INPUT "The length of the hypotenuse? ",hypotenuse

ratio:=nearside/hypotenuse

PRINT \ PRINT "Your angle is ";
PRINT ACS(ratio); " degrees."
PRINT

END
```

See also: **RAD, SIN, COS, TAN, ACS, ASN**

DELETE

INPUT/OUTPUT STATEMENT

SYNTAX: DELETE <STRING expression>

DELETE does not return a value. **DELETE** is used to remove files from a directory. The file is destroyed.

DELETE needs one argument — a string expression.

EXAMPLE::

```
PROCEDURE deletedem
(* Show DELETE statement in action *)
(* We'll Create a file, write something *)
(* to it, read it and finally, delete it *)

DIM file:INTEGER

PRINT
CREATE #file,"demofile":UPDATE
PRINT #file,"This message is going into a dummy file."
CLOSE #file

PRINT "We have created a file and written a sentence in it."
PRINT "We'll do a DIR — so you may look for it."
PRINT "It's named, 'demofile'."
PRINT

SHELL "dir"
PRINT

PRINT \ PRINT "Now let's delete 'demofile'."

PRINT "And, prove that it is gone."
SHELL "del demofile"
SHELL "dir"
```

See also: **CREATE**

DECLARATIVE STATEMENT

```
SYNTAX: DIM <decl seq> {;<decl seq>}
            <decl seq>:= <decl> {,<decl>}[:<type>]
               <decl>:= <name>[<subscript>]
          <subscript>:=(<const>[,<const>[;<const>]])
               <type>:=BYTE|INTEGER|REAL|BOOLEAN|
                       STRING|STRING<max len>|<user defined>
       <user defined>:=user defined by TYPE statement
```

DIM declares variables, arrays, and complex data structures.

DIM may have one or several arguments. Several data types may be declared with this statement. **DIM** reserves memory space for variables, arrays and complex data types.

EXAMPLE:

```
PROCEDURE dimdim
DIM number:BYTE
DIM numbertwo:INTEGER
DIM numberthree:REAL
DIM judgement:BOCIEAN
DIM word:STRING[4]


PRINT
number:=255 \ PRINT number
number:=256 \ PRINT number
number:=257 \ PRINT number
PRINT

numbertwo:=65535. \ PRINT numbertwo
numbertwo:=45.78 \ PRINT numbertwo

PRINT
numberthree:=45.78 \ PRINT numberthree
numberthree:=45.78**8 \ PRINT numberthree
PRINT

judgement:=numberthree>numbertwo
PRINT judgement
PRINT

word:="Sometimes"
PRINT word
PRINT
```

See also: **PARAM, TYPE, BOOLEAN, BYTE, INTEGER, REAL, STRING**

SYSTEM MODE COMMAND

SYNTAX DIR [<pathlist>]

DIR displays the name, size, variable storage requirement, and type of each procedure present in BASIC09's workspace. It also returns the amount of free memory.

DIR accepts an optional argument — a pathlist to a file where you would like to store the listing.

See also: **CHD, KILL, LOAD, MEM**

CONTROL STATEMENT

SYNTAX: END [<output list>]

END stops your procedure and returns to the calling procedure or to the BASIC09 Command Mode.

END may have an optional argument — an output list that you would like **PRINT**ed on the standard output device.

END may be used more than once in a procedure. It does not have to be located at the bottom of a procedure.

EXAMPLE:

```
PROCEDURE endem

(* Show use of END statement *)
(* It marks the end of a procedure *)
(* and tells BASIC09 to return. *)
(* Notice that it also lets you print *)
(* a message when you exit the procedure *)

DIM loopnumber:INTEGER

loopnumber:=0

WHILE loopnumber<5 DO
loopnumber:=loopnumber+1
PRINT "Loop # "; loopnumber

IF loopnumber=3 THEN
PRINT
END "Done after three loops!"
ENDIF

ENDWHILE
END
```

See also: **BYE, ERROR, PAUSE, STOP**

BOOLEAN FUNCTION

SYNTAX: EOF (<NUM>)

EOF returns a **TRUE** or **FALSE** value.

EOF needs one parameter — a path number.

EOF is used to test for an end of file condition which means that all the data on the file has been read. It should be used **AFTER** a **READ** or **GET** statement. A **TRUE** value is returned as long as data is available from the path named.

EXAMPLE:

```
PROCEDURE eofdem
(* Show use of EOF test *)
(* Create a file, read it a character *)
(* at a time — until you reach the end of file *)

DIM char:STRING[1]
DIM file:INTEGER

CREATE #file,"demofile":UPDATE
PRINT #file,"This is a short file."
CLOSE #file

PRINT
PRINT "We have created a short file. "
PRINT "Now, we'll read from it a character at a time "
PRINT "— until we run out of characters."
PRINT

OPEN #file,"demofile":READ

WHILE NOT(EOF(#file)) DO
GET #file,char
PUT #1,char
ENDWHILE

CLOSE #file

PRINT \ PRINT
PRINT "You have reached the end of the file, 'demofile'."
PRINT

SHELL "del demofile"
```

See also: **READ, GET, TRUE, FALSE**

CONTROL STATEMENT

SYNTAX: ERROR (<INT EXPR>)

ERROR is used to intentionally cause the computer to process an error with the error code set equal to the number in the integer expression.

ERROR needs one argument — an integer expression to be used as the desired error code.

This statement is used to test error handling routines or to terminate programs that need to return error status to a calling program.

EXAMPLE:

```
PROCEDURE errordem
(* Show use of ERROR statement *)
(* to generate an error code for testing *)

DIM errornumber:INTEGER

ON ERROR GOTO 10

PRINT
PRINT "Now is the time for all good men ..."
ERROR 211
PRINT "to come to the aid of their country."

10 errornumber:=ERR

PRINT
PRINT "You have encountered Error Number "; errornumber;
PRINT "."
PRINT "This error condition was caused by the ERROR statement"
PRINT "in the program."
PRINT
```

See also: **ON ERROR GOTO, ERR**

FUNCTION

SYNTAX: ERR

ERR returns the **INTEGER** value of the most recent error's error code.

ERR needs no parameters.

ERR is used to determine or display the most recent error. **ERR** returns the type of error by error code number and automatically resets to zero after execution.

EXAMPLE:

```
PROCEDURE errdem
(* Show use of ERR numbers *)
(* Create a file, read it a character *)
(* at a time — until you reach the end of file *)
DIM char:STRING[1]
DIM count,file,errornumber:INTEGER
CREATE #file,"demofile":UPDATE
PRINT #file,"This is a short file."
CLOSE #file
ON ERROR GOTO 10
PRINT
PRINT "We have created a short file. "
PRINT "Now, we'll read from it a character at a time "
PRINT "— until we run out of characters."
PRINT
OPEN #file,"demofile":READ
FOR count:=1 TO 100
GET #file,char
PUT #1,char
NEXT count
10 errornumber:=ERR
PRINT \ PRINT
PRINT "You have encountered ERROR Number "; errornumber
CLOSE #file
PRINT
PRINT "This means you have reached the end of the file."
PRINT
SHELL "del demofile"
```

See also: **ERR**

CONTROL STATEMENT

```
SYNTAX: EXITIF <BOOLEAN EXPR> THEN <statements>
        <statements>
        ENDEXIT
```

EXITIF provides an exit test at any place within any kind of Basic09 loop or control structure. It also provides for a sequence of statements to be executed if the exit is taken.

If the **BOOLEAN** value of the expression following **EXITIF** is **TRUE**, the statements up to the **ENDEXIT** are executed, then the current loop is exited.

If the expression is **FALSE**, the statement following the **ENDEXIT** is executed next, and the current loop level is maintained.

EXAMPLE:

```
PROCEDURE exitifdem
(* Show use of the EXITIF ... THEN ... ENDEXIT clause *)

DIM escapeloopnumber,loopcount:INTEGER

loopcount:=0
escapeloopnumber:=185

LOOP

loopcount:=loopcount+1

EXITIF escapeloopnumber=loopcount THEN
PRINT
PRINT "The counters are equal — It's time to stop!"
ENDEXIT

PRINT ".";
ENDLOOP
```

See also: **FOR..NEXT, REPEAT..UNTIL, LOOP..ENDLOOP, WHILE..DO**

EXP

MATH FUNCTION

SYNTAX: EXP(<NUM>)

EXP returns the base value of the natural logarithm e (2.71828183) raised to the power <**NUM**>.

EXP needs one parameter — any expression that gives a numeric result.

EXP is the opposite of the math function **LOG**.

EXAMPLE:

```
PROCEDURE expdem
(* Show the use of the EXP function *)

DIM number,exponent:REAL

number: = 4.6051705
exponent: = EXP(number)

PRINT "The natural exponential of "; number;
PRINT " is "; exponent
```

See also: **LOG, LOG10**

FALSE

BOOLEAN FUNCTION

SYNTAX: FALSE

FALSE returns the **BOOLEAN** value **FALSE**. It is usually used to assign a value to a **BOOLEAN** variable.

FALSE has no arguments.

EXAMPLE:

```
PROCEDURE falsedem
(* Show use of FALSE function *)
```

```
DIM status:BOOLEAN

status: = FALSE

IF status THEN
PRINT "It must be true!"
ELSE
PRINT "It sure is false.!"
ENDIF
```

See also: **BOOLEAN, TRUE**

MATH FUNCTION

SYNTAX: FIX(<NUM>)

FIX returns the value of <**NUM**> as an **INTEGER**. It removes all numbers to the right of the decimal point by truncating.

FIX needs one parameter — a numeric expression.

FIX is used to convert real numbers or convert values for use in other logical or math functions where an **INTEGER** is required. **FIX** is the opposite of the math function **FLOAT**. Don't confuse **FIX** with **INT**. **INT** rounds real numbers; **FIX** converts **REAL** type numbers to **INTEGER** type numbers.

EXAMPLE:

```
PROCEDURE fixdem
(* Show FIX function in action *)

DIM number,numbertwo:REAL

number: = 123.4567
numbertwo: = FIX(number)

PRINT "If you print "; number;
PRINT " after FIXing it, you will see "; numbertwo
```

See also: **FLOAT**

MATH FUNCTION

SYNTAX: FLOAT(<NUM>)

FLOAT returns the value of <**NUM**> as a **REAL** number.

FLOAT needs one parameter — a numeric expression.

FLOAT is used to convert a **BYTE** or **INTEGER** type value to type **REAL**. This is used to increase the accuracy of the result of a numeric expression or to make a **REAL** value where a **REAL** is required. It is the opposite of the math function **FIX**.

EXAMPLE:

```
PROCEDURE floatdem
(* Demonstrate use of FLOAT function *)

DIM number:INTEGER
DIM result:REAL

number: = 258
result: = FLOAT(number)

PRINT
PRINT "If you print "; number;
PRINT " after using it in the FLOAT function,"
PRINT "you will see a REAL number: "; result
PRINT
PRINT "Notice that BASIC09 always prints REAL numbers"
PRINT "with a decimal point."
PRINT
```

See also: **FIX**

CONTROL STATEMENT

SYNTAX: FOR <VAR> = <EXPR> TO <EXPR> [STEP <EXPR>]
<statements>
NEXT

FOR is the first part of the **FOR** ... **TO** ... **NEXT** statement and is used to assign numbers to numeric variables within the range specified by **FOR** and **TO**.

TO tells BASIC09 how many times to execute a loop. The expression on either side of **TO** may have an **INTEGER** or a **REAL** value. However, a **FOR** ... **TO** ... **NEXT** loop that uses **INTEGER** counters is much faster.

NEXT causes the counter variable to be automatically increased or decreased. It tells BASIC09 to return and execute the statement following the **FOR** part of the statement.

STEP determines the size of the increase or decrease. If no **STEP** is indicated in the statement, BASIC09 automatically uses a **STEP** of one.

EXAMPLE:

```
PROCEDURE fordem
(* Show FOR ... NEXT construct at work *)

DIM count:INTEGER

FOR count:=1 TO 5
PRINT count;
PRINT " ";
NEXT count

PRINT

FOR count:=5 TO 100 STEP 5
PRINT count;
PRINT " ";
NEXT count

PRINT
```

See also: **none**

INPUT/OUTPUT STATEMENT

SYNTAX: GET <EXPR>,<struct name>

GET is used to read binary data records from a file or device.

GET needs two arguments. The first is an expression that evaluates to the number of the input/output path. The second is the name of a variable, array or complex data structure.

GET is commonly used to read elements of a random access file. It is also used to read single, raw characters from a terminal.

EXAMPLE:

```
PROCEDURE getdem
(* Show use of GET statement *)
(* Create a file, read in a character *)
(* at a time — until you reach the end of file *)

DIM char:STRING[1]
DIM file:INTEGER

CREATE #file,"demofile":UPDATE
PRINT #file,"This is another test file."
CLOSE #file

PRINT
PRINT "We have created a short file. "
PRINT "Now, we'll read from it a character at a time "
PRINT "using the GET statement to fetch a string with one character."
PRINT

OPEN #file,"demofile":READ

WHILE NOT(EOF(file)) DO
GET #file,char
PUT #1,char
ENDWHILE

PRINT \ PRINT
CLOSE #file

SHELL "del demofile"
```

See also: **CREATE, PUT**

CONTROL STATEMENT

SYNTAX: GOSUB <line >

GOSUB is the first part of the **GOSUB** ... **RETURN** control construct. **GOSUB** transfers control of program flow to a subroutine.

GOSUB needs one argument — a line number. When BASIC09 finds a **RETURN** in the subroutine, it returns and executes the statement following **GOSUB**.

GOSUB is used to branch out of a procedure's main flow to a subroutine. It may be used to link several separate subroutines into a main procedure.

EXAMPLE:

```
PROCEDURE gosubdem
(* Show a subroutine in use *)
DIM count:INTEGER

PRINT

FOR count: = 1 TO 9
GOSUB 10
NEXT count

STOP

10 PRINT count;
PRINT " ";
PRINT "Hello Again."
RETURN
```

See also: **ON, GOTO**

CONTROL STATEMENT

SYNTAX: GOTO <line >

GOTO does not return a value. It causes a jump to the line number you specify.

GOTO needs one argument — a line number or a variable that represents a line number.

GOTO is used to force your procedure to branch to a new line instead of executing the next line in sequence.

EXAMPLE:

```
PROCEDURE gotodem
10 REM Show use of GOTO statement
20 PRINT . "The GOTO statement ";
30 GOTO 60
40 PRINT "doesn't work."
50 STOP
60 PRINT "works well but is a dangerous"
70 PRINT "programming practice."
80 END
```

See also: **GOSUB, ON**

CONTROL STATEMENT

SYNTAX: IF <BOOLEAN expression> THEN <line #>

OR: IF <BOOLEAN expression> THEN <statements>
[ELSE <statements>]
ENDIF

IF is the first part of the **IF** ... **THEN** ... **ELSE** control construct. It does not return a value. It indicates a variable is to be tested using a **BOOLEAN** expression such as one of BASIC09's relational operators. If the condition tested is **TRUE**, control is transferred to the line number following **THEN**.

If no line number is used, you must use the second form of the statement. Then, if the condition tested is **TRUE**, the statements between the **THEN** and the **ENDIF** are executed.

ELSE causes BASIC09 to skip all statements between **ELSE** and **ENDIF** when the **BOOLEAN** expression following the **IF** statement is **TRUE**.

ELSE diverts the flow of a procedure to the alternate statements between **ELSE** and **ENDIF** when the expression following the **IF** statement is **FALSE**.

EXAMPLE:

```
PROCEDURE ifdem
(* Demonstrate use of the IF ... THEN ... ELSE ... ENDIF *)

DIM numbone,numbtwo:INTEGER

numbone:=333
numbtwo:=222

PRINT
PRINT "The first number is: "; numbone
PRINT "The second number is: "; numbtwo
PRINT

IF numbone<numbtwo THEN
PRINT "The first number is smaller! "
ELSE

PRINT "The second number is smaller."

ENDIF

END
```

See also: **none**

INPUT/OUTPUT STATEMENT

SYNTAX: INPUT [<INT EXPR>,]["<prompt>",] <input list>

INPUT reads values from a terminal and assigns them to variable(s). However, it returns an error message if you enter the wrong **TYPE** of data. It causes the computer to print a question mark when it is waiting for you to **INPUT** data.

INPUT needs a variable name or a list of variable names that tell BASIC09 where to store data being **INPUT**. It also accepts two optional arguments; a path number in the form of an **INTEGER** expression and a prompt for display on your terminal.

INPUT is used to assign values to variables during program execution. The values normally come from your terminal, but will be fetched from an external storage device if a path number is specified.

EXAMPLE:

```
PROCEDURE inputdem
(* Show INPUT statement in action *)
DIM age,path:INTEGER
DIM name:STRING[18]
DIM sentence:STRING[80]
(* First from the keyboard *)
INPUT "What is your name? ",name
PRINT "What is your age "; name;
INPUT age
PRINT
PRINT "Nice to meet you "; name;
PRINT ". I'm glad you are "; age; "."
PRINT
PRINT "Excuse me for a minute while I create a file "
PRINT "and read a sentence from it."
PRINT
(* Then from a file *)
(* First we must create a file to read *)
CREATE #path,"demofile":WRITE
PRINT #path,"I was read from a file named 'demofile'."
CLOSE #path
(* Now, we'll open it for read *)
(* and print the data *)
OPEN #path,"demofile":READ
INPUT #path,sentence
CLOSE #path
SHELL "del demofile"
PRINT sentence
PRINT
```

See also: **GET, READ**

MATH FUNCTION

SYNTAX: INT<num>

INT rounds a value to the nearest whole number. It returns a **REAL** value.

INT needs one parameter, a **REAL** number. The number may be negative or positive.

Don't confuse **INT** with **FIX**. **INT** rounds real numbers; **FIX** converts **REAL** type numbers to **INTEGER** type numbers.

EXAMPLE:

```
PROCEDURE intdem
(* Show INT function *)
DIM number,roundednumber:REAL
DIM count,integernumber:INTEGER

DATA 3.336,5.761,101.888

PRINT
PRINT "First we'll print a number and its INTeger value."
PRINT

FOR count:=1 TO 3
READ number
integernumber:=INT(number)
PRINT number,integernumber
NEXT count

PRINT
PRINT "Then, we'll print the same numbers as if they were"
PRINT "money rounded off to the nearest whole penny."
PRINT

(* Now let's use INT to round off the same numbers *)

FOR count:=1 TO 3
READ number
number:=number*100+.5
roundednumber:=INT(number)
roundednumber:=roundednumber/100
PRINT roundednumber
NEXT count

PRINT
```

INTEGER

TYPE SPECIFICATION

SYNTAX: DIM <VARIABLE>: INTEGER

INTEGER TYPEs a variable.

INTEGER needs one argument; a single variable or list of variables.

INTEGER is used with the **DIM** or **PARAM** statement to **TYPE** variables and reserve memory for them. **INTEGER** variables may never store a value less than -32768 or greater than 32767. Arithmetic with **INTEGER** variables is much faster than arithmetic with **REAL** variables.

See also: **BYTE, REAL, STRING, BOOLEAN**

KILL

CONTROL STATEMENT

SYNTAX: KILL <STR EXPR>

KILL deletes external procedures from BASIC09's directory.

KILL needs one argument; a string expression that names an external procedure.

KILL is used to free up system memory. It will not delete a procedure located in BASIC09's workspace.

EXAMPLE:

```
PROCEDURE killdem
(* Show how you can remove a packed procedure *)
(* with BASIC09's KILL statement *)
DIM favoriteprocedure:STRING[31]
PRINT
PRINT "WARNING: You must ask for a packed BASIC09 procedure"
PRINT "that you have already loaded into memory or one that"
PRINT "is stored in your current execution directory."
PRINT "Otherwise you will cause an 'unknown procedure' error."
PRINT
INPUT "Which procedure would you like to run? ",favoriteprocedure
RUN favoriteprocedure
KILL favoriteprocedure
SHELL "mdir"
```

```
PRINT "Notice that the KILL statement has removed your"
PRINT "procedure from memory."
PRINT
```
See also: **none**

KILL

SYSTEM MODE COMMAND

SYNTAX KILL <PROCEDURE NAME>
KILL*

KILL erases a procedure (**KILL***, all procedures) from the BASIC09 workspace.

KILL needs one argument, the name(s) of the procedures to be erased.

KILL removes procedures no longer wanted. You must **SAVE** the procedure before **KILL**ing or it will be permanently erased.

LAND

LOGICAL FUNCTION

SYNTAX: LAND(<num>,<num>)

LAND returns an **INTEGER** result.

LAND needs two parameters of type **INTEGER** or **BYTE**.

LAND performs logical operations bit by bit. It returns an **INTEGER** result. **LAND** stands for Logical **AND**. It is not a **BOOLEAN** operator.

EXAMPLE:
```
PROCEDURE landdem
(* Show Logical AND function *)
DIM number,numbertwo:BYTE
PRINT "1 = 00000001"
PRINT "3 = 00000011"
PRINT
PRINT "The Logical AND should be '1' or 00000001"
PRINT "Let's try it."
PRINT
number:=1
numbertwo:=3
PRINT LAND(number,numbertwo)
PRINT
```
See also: **LOR, LXOR, LNOT**

LEFT$

STRING FUNCTION

SYNTAX: LEFT$(<string$>,<INT>)

LEFT$ returns a specified number of characters from the left end of a **STRING** variable.

LEFT$ needs two parameters. The first must be a **STRING** constant or the name of a **STRING** variable. The second must be an **INTEGER** number with a value between zero and 255 decimal.

LEFT$ is often used in **PRINT** statements. It may also be used to assign a value to another **STRING** variable.

EXAMPLE:

```
PROCEDURE leftdem
(* Demonstrate LEFT$ Statement *)
DIM examplestring:STRING
examplestring:="REMarkable!"
PRINT "Who made that "; LEFT$(examplestring,6); "?"
END
```

See also: **CHR$, MID$, RIGHT$, TRIM$, DATE$**

LEN

STRING FUNCTION

SYNTAX: LEN(<STRING>)

LEN returns an **INTEGER** or **BYTE** value.

LEN needs one parameter, a literal **STRING** or **STRING** variable.

LEN determines the length of a **STRING** variable by counting the number of characters enclosed in quotes or assigned to a string variable.

EXAMPLE:

```
PROCEDURE lendem
(* Show how to use the LEN function *)
DIM sentence:STRING[80]
PRINT
PRINT "Type a few words.  Take up to 80 characters, but don't"
PRINT "use any commas."
PRINT
```

```
INPUT "Your words: ",sentence
PRINT
PRINT "You typed, '"; sentence; "' — ";
PRINT "a total of "; LEN(sentence); " characters."
PRINT
```

See also: **ASC, SUBSTR**

LET

ASSIGNMENT STATEMENT

```
SYNTAX: [LET] <var> := <EXPR>
        [LET] <var> = <EXPR>
        [LET] <struct> := <struct>
        [LET] <struct> = <struct>
```

LET assigns values to variables. The **LET** keyword is optional; an assignment by itself does the same thing.

LET needs two arguments. One must be a variable and the other an expression. The second may also be the value of an array or data structure that you want to copy into another complex structure.

LET evaluates an expression and stores the result in a variable. It can also be used to copy complex data structures.

EXAMPLE:

```
PROCEDURE letdem
(* Show use of LET statement *)
DIM number:INTEGER
LET number:=45
PRINT
PRINT number
PRINT
PRINT "The LET statement let you set the value of 'number'"
PRINT "to 45 in the code above. But, it is an optional statement."
PRINT
PRINT "'number := 56' — works just as well."
number:=56
PRINT
PRINT number
PRINT \ PRINT "See!"
PRINT
```

See also: **none**

LIST

SYSTEM COMMAND

SYNTAX: LIST

LIST displays a formatted listing of the current procedure.

LIST needs no arguments.

LIST may be used from both the system mode and the **DEBUG** mode. If the optional asterisk is used, **LIST*** lists all procedures in your workspace.

See also: **BREAK, CONT, DIR, END, STATE**

LNOT

LOGICAL FUNCTION

SYNTAX: LNOT(<num>)

LNOT returns an **INTEGER** result.

LNOT needs one parameter, an **INTEGER** or **BYTE** number.

LNOT performs logical operations bit by bit and returns **INTEGER** results. **LNOT** stands for Logical **NOT**. **IT** is not a **BOOLEAN** operator.

EXAMPLE:

```
PROCEDURE lnotdem
(* Show how Logical NOT statement is used *)

DIM number,numbernot:INTEGER

PRINT
INPUT "Type a number between -32768 and 32767: ",number

numbernot:=LNOT(number)

PRINT "The binary complement of "; number; " is ";
PRINT numbernot
PRINT
```

See also: **LAND, LOR, LXOR**

SYSTEM COMMAND

SYNTAX: LOAD <pathlist>

LOAD loads procedures into the computer from a disk or tape file.

LOAD needs one argument. It may be the name of a file in the current data directory, a complete OS-9 pathlist or the name of a device.

LOAD is used to load procedures into BASIC09's workspace. It is entered from the System Mode.

See also: **CHD, EDIT, KILL, LIST, RUN, SAVE**

MATH FUNCTION

SYNTAX: LOG(<num>)

LOG returns the natural logarithm of a number greater than zero.

LOG needs one parameter, a **REAL** or **INTEGER** number or a variable of those **TYPE**s.

LOG finds the power a number must be raised to in order to get a known result.

EXAMPLE:

```
PROCEDURE logdem
(* Show LOG function in use *)

DIM number:REAL

PRINT
INPUT "Type a positive number: ",number
PRINT

PRINT "The natural log of "; number;
PRINT " is "; LOG(number)
PRINT
```

See also: **EXP, LOG10**

LOG10

MATH FUNCTION

SYNTAX: LOG10(<num>)

LOG10 returns a **REAL** number. It is the opposite of **LOG**.

LOG10 needs one parameter, a **REAL** or **INTEGER** number or variable.

LOG10 finds the value of the "common" or base 10 logarithm of any number whose value is greater than zero.

EXAMPLE:

```
PROCEDURE log10dem
(* Demonstrate use of LOG10 function *)
DIM number:REAL
PRINT
INPUT "Type a positive number: ",number
PRINT
PRINT "The common log of "; number;
PRINT " is "; LOG10(number)
PRINT
```

See also: **EXP, LOG**

LOOP ... ENDLOOP

CONTROL STATEMENT

```
SYNTAX: LOOP
        ENDLOOP
```

LOOP is part of the **LOOP** ... **ENDLOOP** construct.

LOOP defines the start of a loop. Statements between **LOOP** and **ENDLOOP** are executed over and over again. They will **RUN** forever unless an **EXITIF** statement is used to create an exit condition..

ENDLOOP defines the end of a loop. When BASIC09 sees this word it returns to the statement following the word **LOOP** and starts over.

EXAMPLE:

```
PROCEDURE loopdem
(* Show the LOOP ... ENDLOOP Construct *)
(* Dangerous: It will loop forever without EXITIF clause *)
DIM countlessloops:INTEGER
```

```
countlessloops: = 0
LOOP
countlessloops: = countlessloops + 1
PRINT "* ";
EXITIF countlessloops > 150 THEN
PRINT
PRINT "That's 150 stars!"
ENDEXIT
ENDLOOP
```

See also: **ENDLOOP, EXITIF**

LOGICAL FUNCTION

SYNTAX: LOR(<num>,<num>)

LOR returns an **INTEGER** result.

LOR needs two parameters, of type **INTEGER** or **BYTE**.

LOR stands for Logical **OR**. It performs bit by bit logical operations. **LOR** is not a **BOOLEAN** operator.

EXAMPLE:

```
PROCEDURE lordem
(* Show Logical OR — LOR *)

DIM number,numbertwo:INTEGER

PRINT
PRINT "3 = 00000011"
PRINT "8 = 00001000"
PRINT
PRINT "The Logical OR of '3' and '8' should be '11'."
PRINT "Let's try it."

number: = 3
numbertwo: = 8

PRINT
PRINT LOR(number,numbertwo)
PRINT
PRINT "How about that!"
PRINT
```

See also: **LAND, LNOT, LXOR**

LXOR

LOGICAL FUNCTION

SYNTAX: LXOR(<num>,<num>)

LXOR returns an **INTEGER** result.

LXOR needs two parameters, of type **INTEGER** or **BYTE**.

LXOR is used to perform a bit by bit Exclusive **OR** operation on **BYTE** or **INTEGER** data. It stands for **L**ogical e**X**clusive **OR**.

EXAMPLE:

```
PROCEDURE lxordem
(* Show Logical eXclusive OR *)

DIM number,numbertwo:INTEGER

PRINT
PRINT "3 = 00000011"
PRINT "9 = 00001001"
PRINT
PRINT "The Logical eXclusive OR of '3' and '9' should be '10'."
PRINT "Let's try it."

number:=3
numbertwo:=9

PRINT
PRINT LXOR(number,numbertwo)
PRINT
PRINT "Just like magic!"
PRINT
```

See also: **LAND, LOR, LNOT**

MEM

SYSTEM MODE COMMAND

SYNTAX: MEM
MEM <number>

MEM returns a decimal number which is the amount of available memory in the BASIC09 workspace.

MEM accepts an optional argument which may be a hexadecimal number or a decimal number.

MEM is used to find out how much room is left in the workspace. However, it may also be used to expand the workspace. In this case the number of bytes of memory needed is typed following **MEM**.

See also: **DIR, KILL, LOAD,**

STRING FUNCTION

SYNTAX: MID$(<string$>,<int1>,<int2>)

MID$ returns the requested number of characters from the middle of a **STRING** variable.

MID$ needs three parameters. The first is the name of the **STRING** (or a **STRING** expression) the characters are to be taken from. The second is an **INTEGER** value that marks the location of the first character to be removed. The third is an **INTEGER** value that tells BASIC09 how many characters to take.

MID$ is used to isolate a specified number of characters from the middle of a **STRING**. It may also be used to assign a value to another STRING variable.

EXAMPLE:

```
PROCEDURE middem
(* Demonstrate the use of MID$ Statement *)

DIM examplestring:STRING

examplestring:="REMarkable!"

PRINT "That is the "; MID$(examplestring,3,4);
PRINT " of an "; MID$(examplestring,7,4); "" man."

END
```

See also: **CHR$, LEFT$, RIGHT$, TRIM$, DATE$**

MATH FUNCTION

SYNTAX: MOD(<num1>,<num2>)

MOD returns a number of type **INTEGER** or **BYTE**.

MOD needs two parameters of type **INTEGER** or **BYTE**.

MOD computes the arithmetic remainder after a division (the modulus function).

EXAMPLE:

```
PROCEDURE moddem
(* Show MOD function *)

DIM dividend,divisor,quotient:REAL
DIM integerresult,resultofmod:INTEGER

PRINT
PRINT "Let's divide one number by another."
PRINT
INPUT "First, give me a number to be divided: ",dividend
INPUT "Now, give me the number you wish to divide it by: ",divisor

quotient:=dividend/divisor
integerresult:=INT(quotient)
resultofmod:=MOD(dividend,divisor)

PRINT
PRINT dividend; " divided by "; divisor; " is "; quotient
PRINT
PRINT "Or, it is "; integerresult; " with a remainder"
PRINT "— or MODulo — of "; resultofmod
PRINT
```

See also: **ABS, FIX**

BOOLEAN FUNCTION

SYNTAX: NOT <expression>

NOT returns a **TRUE** or **FALSE** value. It is most often used in the **IF** ... **THEN** conditional statement.

NOT needs one parameter, an expression that evaluates to **TRUE** or **FALSE.**

NOT is used as a logical operator to reverse a normal result.

For example: If "**ALPHA > BETA**" is **TRUE, "NOT(ALPHA > BETA)**" would be **FALSE**.

EXAMPLE:

```
PROCEDURE notdem
(* Show Boolean NOT operator *)
DIM status:BOOLEAN
DIM firstscore,secondscore:INTEGER

PRINT
INPUT "What was your score in the first game? ",firstscore
INPUT "What was your score in the second game? ",secondscore
PRINT

status:=firstscore>secondscore

IF NOT(status) THEN
PRINT "It looks like the first game was a warm-up."
ELSE
PRINT "It looks like you should have quit while you were ahead."
ENDIF

PRINT
```

See also: **AND, OR, XOR**

CONTROL STATEMENT

SYNTAX: ON ERROR [GOTO <line >]

ON ERROR GOTO transfers control to the specified line when an error occurs.

ON ERROR GOTO needs one argument — a line number that tells your program where to **GO** when an error occurs. You may use **ON ERROR** without the line number to force your procedure to enter the **DEBUG** mode when an error occurs.

ON ERROR GOTO may be used to "trap" program errors and report them. This does not happen until after the **ON ERROR GOTO** is executed. Another **ON ERROR GOTO** can be used later to change the error routine.

EXAMPLE:

```
PROCEDURE onerrordem
(* Show ON ERROR GOTO in use *)
DIM number,result:REAL

ON ERROR GOTO 10

PRINT "Type a number and we'll print its inverse. "
PRINT "After you have tried a few numbers try a value of zero."
PRINT
INPUT "Your number: ",number

result:=1/number

PRINT
PRINT "The inverse of "; number; " is "; result
PRINT

STOP

10 PRINT
PRINT "You cannot divide by zero.  Sorry we can't help you."
PRINT "Goodbye!"
PRINT
```

See also: **ERR, ERROR**

CONTROL STATEMENT

SYNTAX: ON <int expr> GOSUB <line > {, <line >}

ON GOSUB is a multiple subroutine branching scheme that accomplishes a number of **IF** ... **GOSUB** tests in one statement.

ON GOSUB needs two arguments. The first, a positive **INTEGER** or the positive **INTEGER** result of an expression. The second, a line number from a list following the verb **GOSUB**.

ON GOSUB may be used with the **INPUT** statement to write menu-driven programs.

EXAMPLE:

```
PROCEDURE ongosubdem
(* Show ON ... GOSUB statement in action *)

DIM card:INTEGER

REPEAT
card:=INT(13*RND(0))+1
card:=card-10
UNTIL card>0 AND card<4

ON card GOSUB 100,110,120

STOP

100 PRINT "You drew a JACK"
RETURN

110 PRINT "You drew a QUEEN"
RETURN

120 PRINT "You drew a KING"
RETURN
```

See also: **ON GOTO**

CONTROL STATEMENT

SYNTAX: ON <int expr> GOTO <line > {,<line >}

ON GOTO is a multiple branching scheme similar to **ON** ... **GOSUB**.

ON GOTO needs two arguments. The first, a positive **INTEGER** or the **INTEGER** result of an expression. The second, a line number from a list following the verb **GOTO**.

ON GOTO may also be used with the **INPUT** statement to write menu-driven programs.

EXAMPLE:

```
    PROCEDURE ongotodem
    (* Show ON ... GOTO statement in action *)

    DIM card:INTEGER

    REPEAT
    card:=INT(13*RND(0))+1
    card:=card-10
    UNTIL card>0 AND card<4

    ON card GOTO 100,110,120

100 PRINT "You drew a JACK"
    STOP

110 PRINT "You drew a QUEEN"
    STOP

120 PRINT "You drew a KING"
    STOP
```

See also: **INPUT, ON GOSUB**

INPUT/OUTPUT STATEMENT

SYNTAX: OPEN <int var>,<string exp> [: <access mode>]
<access mode> := <mode> ! <mode> + <access mode>
<mode> := READ ! WRITE ! UPDATE !
EXEC ! DIR

OPEN opens an input or output path to a file or device.

OPEN needs several arguments. The first, an **INTEGER** number — or an expression or variable that evaluates as an **INTEGER**. This number becomes the input/output path number. A **STRING** expression gives the file name or complete pathlist. You may also specify the access mode — **READ**, **WRITE** or **UPDATE**.

OPEN may be used to access files or devices — even the Execution and Data Directories. See your OS-9 User's Guide for details about file structure.

EXAMPLE:

```
PROCEDURE opendem
(* Show how to use OPEN statement *)

DIM printer:INTEGER
DIM device_name:STRING[2]

device_name:= "/p"

OPEN #printer,device_name:WRITE
PRINT #printer
PRINT #printer,"We have opened an output path to the system printer."
PRINT #printer,"We must remember to CLOSE it when we are through"
PRINT #printer,"so that other users on the system can use it."
PRINT #printer

CLOSE #printer

PRINT "We have sent data to the printer. But, we could"
PRINT "just as easily have sent it to a file on one of"
PRINT "your floppy disks."
PRINT
```

See also: **CLOSE, PRINT, WRITE**

BOOLEAN FUNCTION

SYNTAX: IF <expression> OR <expression> THEN <line>

OR returns a **BOOLEAN** result.

OR needs two parameters. Both expressions must evaluate to a **BOOLEAN** result.

OR is used in an **IF/THEN** statement to test for multiple conditions. If either of the expressions in an **OR** statement evaluates as **TRUE**, control is transferred to the line number or statement following **THEN**.

EXAMPLE:

```
PROCEDURE ordem
(* Demonstrate use of Boolean OR *)

DIM status:BOOLEAN
DIM number:INTEGER

PRINT
INPUT "Type a number between one and 10: ",number

(* status becomes true if you don't follow directons *)

status:=number<1 OR number>10

IF status THEN
PRINT
PRINT "Better follow directons!"
PRINT "Next time type a number between one and 10."
ELSE
PRINT
PRINT "Your number was "; number
PRINT "Thanks!"
ENDIF

PRINT
```

See also: **XOR, AND, NOT**

SYSTEM COMMAND

SYNTAX: PACK[<proc name> {,<proc name> }]
[> <pathlist>]
PACK*[<pathlist>]

PACK does not return a value. **PACK** causes an extra compiler pass that makes your procedure smaller and faster.

PACK needs several arguments; the name of the procedures you want to **PACK** and a pathlist which names the file where they are to be stored.

PACK produces compact procedures that load and run in memory outside BASIC09's workspace. **PACK**ed procedures cannot be edited or debugged. You must always **SAVE** your procedures in source form before performing a **PACK**.

See also: **SAVE**

DECLARATIVE STATEMENT

SYNTAX:	**<decl seq>**	**{; <decl seq> }**
	<decl seq>	**:= <decl> {, <decl> } [: <type>]**
	<decl>	**: <name> =[<subscript>]**
	<subscript>:=	**(<const> [,<const> [,<const>]])**
	<type>	**:= BYTE | INTEGER | REAL | BOOLEAN| STRING | STRING <max len> | <user defined type>**
	<user def>	**:= user defined by TYPE statement**

PARAM needs two arguments — a list of variables, arrays or user defined data types being **DIM**ensioned and the data **TYPE**.

PARAM is similar to the **DIM** statement execpt that it is used to describe the parameters a called procedure should expect to receive from another procedure via a **RUN** statement.

See also: **DIM**

PAUSE

CONTROL STATEMENT

SYNTAX: PAUSE [<output list>]

PAUSE does not return a value.

PAUSE may use an optional argument — a message that is listed on your terminal after the **PAUSE**.

PAUSE is used to insert "breakpoints" in your programs. Execution stops when a procedure that contains a **PAUSE** statement is entered and BASIC09 enters its **DEBUG** mode. You may continue program execution by typing **CONT**.

EXAMPLE:

```
PROCEDURE pausedem
(* demostrate PAUSE statement *)

PRINT "A PAUSE statement causes BASIC09 to enter its Debug Mode."
PRINT "You must type 'cont' to continue."
PRINT
PRINT "You can also let the PAUSE statement print a message to"
PRINT "remind you why the program stopped.  Like this."
PRINT

PAUSE "I quit.  You must type 'cont' to continue."
PRINT
PRINT "Thanks!"
PRINT
```

See also: **CONT**

PEEK

MISCELLANEOUS FUNCTION

SYNTAX: PEEK (<int>)

PEEK returns the **BYTE** value stored at the specified memory address.

PEEK needs one parameter; an **INTEGER** memory address.

PEEK is used to examine memory locations.

EXAMPLE:

```
PROCEDURE peekdem
(* Demonstrate use of PEEK *)
```

```
DIM address,count:INTEGER
DIM sampler:STRING[36]
DIM characterholder:BYTE

(* Assign value to string *)

sampler:="The PEEK function should be X-rated!"

(* Now point directly to STRING in memory *)

address:=ADDR(sampler)

(* Get each character from memory and PRINT it *)

FOR count:=0 TO LEN(sampler)-1
characterholder:=PEEK(address+count)
PRINT CHR$(characterholder);
NEXT count

PRINT

END
```

See also: **POKE**

PI

MATH FUNCTION

SYNTAX: PI

PI returns a **REAL** constant (3.14159265).

PI doesn't need any parameters.

PI is used in expressions that require this constant for calculation.

EXAMPLE:

```
PROCEDURE pidem
(* Show use of PI function *)
DIM circumference,radius:REAL

INPUT "What is the radius of your circle? ",radius

circumference:=2*PI*radius

PRINT
PRINT "Thank you.  The circumference of a circle with a"
PRINT "radius of "; radius; "feet is "; circumference;
PRINT " feet."
PRINT
```

See also: **none**

ASSIGNMENT STATEMENT

SYNTAX: POKE <int expr>, <byte expr>

POKE needs two arguments; an **INTEGER** expression followed by a **BYTE** expression.

POKE stores data of type **BYTE** at a specified address.

EXAMPLE:

```
PROCEDURE pokedem
(* Demonstrate use of POKE *)
DIM address,count:INTEGER
DIM sampler:STRING[10]
DIM characterholder:BYTE

(* Assign value to string *)

sampler:="XXXXXXXXXX"

PRINT
PRINT "Let's print the STRING sampler."
PRINT
PRINT sampler
PRINT

(* Now point directly to 'sampler' in memory *)

address:=ADDR(sampler)

(* And POKE some new characters into the string *)

FOR count:=0 TO LEN(sampler)-1
characterholder:=count+$30
POKE address+count,characterholder
NEXT count

(* Now we'll print the new string *)

PRINT "After the POKE operations, sampler contains: "; sampler
PRINT

END
```

See also: **PEEK**

FUNCTION

SYNTAX: POS

POS returns the position of the cursor or print position in the current **PRINT** line.

POS doesn't need any parameters.

POS is used to find out where the next character will be printed.

EXAMPLE:

```
PROCEDURE posdem
(* Demonstrate a use of the POS function *)
DIM length:INTEGER
DIM example:STRING[80]

PRINT
PRINT "First, we'll print a line of a specified length."
INPUT "How many stars would you like in your line? ",length
PRINT

REPEAT
PRINT "* ";
UNTIL POS>length
PRINT

(* Then we'll use it to count characters *)

PRINT
PRINT "Now, type a line of text — any length. "
PRINT
INPUT "Start here: ",example

PRINT
PRINT example;
length:=POS-1
PRINT
PRINT "You typed "; length; " characters."
PRINT
```

See also: **none**

INPUT/OUTPUT STATEMENT

SYNTAX: PRINT [<int expr>,] <output list>

PRINT sends data to the standard output device — usually your terminal.

PRINT needs an output list containing the data to be printed. It also accepts an **INTEGER** expression that defines the path where the data should be sent.

PRINT is used to output data. It is also often used to display the value of a variable.

See also: **OPEN, CREATE, POS, TAB, USING**

EXAMPLE:

```
PROCEDURE printdem
(* Show use of PRINT statement *)

DIM words(4):STRING[18]
DIM count,numbers(4):INTEGER

DATA 111,112,112,114
DATA "This","is","a","test."

FOR count:=1 TO 4
READ numbers(count)
NEXT count

FOR count:=1 TO 4
READ words(count)
NEXT count
```

```
PRINT
PRINT "First we'll PRINT words and numbers without punctuation."
PRINT

FOR count: = 1 TO 4
PRINT numbers(count)
NEXT count

PRINT

FOR count: = 1 TO 4
PRINT words(count)
NEXT count

PRINT
PRINT "Then we'll print with semicolons."
PRINT

FOR count: = 1 TO 4
PRINT numbers(count);
NEXT count

PRINT
FOR count: = 1 TO 4
PRINT words(count);
NEXT count

PRINT \ PRINT
PRINT "And finally, we'll print with Commas. "
PRINT

FOR count: = 1 TO 4
PRINT numbers(count),
NEXT count

PRINT

FOR count: = 1 TO 4
PRINT words(count),
NEXT count

PRINT \ PRINT
```

See also: **OPEN, CREATE, POS, TAB, USING**

INPUT/OUTPUT STATEMENT

SYNTAX: PUT <expr>,<structure name>

PUT does not return a value.

PUT needs two arguments — an **INTEGER** expression naming a path and a structure or variable name.

PUT writes fixed size binary data to random access files or devices.

EXAMPLE:

```
PROCEDURE putdem
(* Show use of PUT statement *)
(* First, create a file *)

DIM example:STRING[22]
DIM examplefromdisk:STRING[22]
DIM file:INTEGER

example:= "This is a test string."

CREATE #file,"demofile":UPDATE
CLOSE #file

PRINT
PRINT "We have created a short file. "
PRINT "Now, we'll PUT an entire STRING into it at one time."
PRINT

OPEN #file,"demofile":WRITE
PUT #file,example
CLOSE #file

PRINT
PRINT "Let's read our file back now."
PRINT

OPEN #file,"demofile":READ
GET #file,examplefromdisk
PRINT examplefromdisk
PRINT
CLOSE #file

SHELL "del demofile"
```

See also: **CREATE, GET**

DIRECTIVE STATEMENT

SYNTAX: RAD

RAD doesn't require any arguments.

RAD tells a procedure that all angles should be evaluated as **RAD**ians instead of **DEG**rees. **RAD** is the opposite of **DEG**.

EXAMPLE:

```
PROCEDURE raddem
(* Demonstrate use of RAD statement *)

(* Use Radians *)

RAD

DIM nearside,hypotenuse,ratio:REAL

INPUT "The length of the side adjacent to the angle? ",nearside
INPUT "The length of the hypotenuse? ",hypotenuse

ratio:=nearside/hypotenuse

PRINT \ PRINT "Your angle is ";
PRINT ACS(ratio); " radians."
PRINT

END
```

See also: **DEG**

INPUT/OUTPUT STATEMENT

SYNTAX: READ <int expr>,<input list>

READ returns ASCII data from a file or device.

READ needs several arguments; an **INTEGER** expression naming a path number and an input list that tells BASIC09 where to store the data it **READ**s.

READ is used to extract data from files.

EXAMPLE:

```
PROCEDURE readdem
(* Show use of READ statement *)

DIM words(3):STRING[10]
DIM count:INTEGER

DATA "I","Quit!"

(* First read and print internal data *)

PRINT

FOR count:=1 TO 2
READ words(count)
PRINT words(count)
NEXT count
PRINT

(* Now READ from an external source, like the keyboard *)

PRINT "Type three words.  Follow each with a carriage return."
PRINT

FOR count:=1 TO 3
READ #0,words(count)
NEXT count

PRINT
FOR count:=1 TO 3
PRINT words(count)
NEXT count
PRINT
```

See also: **CREATE, GET, OPEN, PRINT, WRITE**

DIRECTIVE STATEMENT

SYNTAX: REM <chars>
(* <chars>[*)]

REM needs one argument — a comment line.

REM is used to insert comments in your program. The character "!" may be typed in place of **REM** when writing procedures. The optional "(*" syntax is compatible with **PASCAL** programs.

See also: none

SYSTEM MODE COMMAND

SYNTAX: RENAME <proc name>,<new proc name>

RENAME needs two arguments — the current procedure name and the new procedure name.

RENAME changes the name of a procedure.

See also: none

REPEAT

CONTROL STATEMENT

SYNTAX: REPEAT
UNTIL <boolean expr>

REPEAT is part of the **REPEAT** ... **UNTIL** loop construct.

REPEAT needs one argument — a boolean expression.

REPEAT is used to execute statements inside a loop **UNTIL** the **BOOLEAN** expression at the end of the loop becomes **TRUE**.

EXAMPLE:

```
PROCEDURE repeatdem
(* Demonstrate REPEAT ... UNTIL Control Structure *)
DIM count:INTEGER

count:=0

PRINT

REPEAT
PRINT count
count:=count+1
UNTIL count>10

PRINT
```

See also: none

RESTORE

INPUT/OUTPUT STATEMENT

SYNTAX: RESTORE [<line number>]

RESTORE does not return a value.

RESTORE needs one argument — a line number.

RESTORE tells BASIC09 which **DATA** statements are to be **READ** next. If a line number is used, that line is **READ** next. If a line number is not used, the first **DATA** statement is read.

EXAMPLE:

```
PROCEDURE restoredem
(* Show use of RESTORE statement *)
DIM count,numbers:INTEGER

DATA 10,20,30,40
DATA 50,60,70,80
10 DATA 90,100,110,120
DATA 130,140,150,160

(* PRINT first 16 data elements *)

PRINT

FOR count:=1 TO 16
READ numbers
PRINT numbers; " ";
NEXT count

PRINT \ PRINT

(* Now read and print last eight elements again *)

RESTORE 10

FOR count:=1 TO 8
READ numbers
PRINT numbers; " ";
NEXT count

PRINT \ PRINT
```

See also: **DATA, READ**

RND

MATH FUNCTION

SYNTAX: RND (<num>)

RND returns a random **REAL** number.

RND needs one parameter — a **REAL** number.

RND may be used to generate random numbers for games or simulations.

See also: none

RUN

**SYSTEM MODE COMMAND,
CONTROL STATEMENT**

**SYNTAX: RUN <proc name> [(<param> {,<param>})]
RUN <string var> [(<param> {,<param>})]**

RUN executes the procedure named.

RUN can have several arguments. The first, a procedure name or the name of a **STRING** variable which holds the procedure name. The second, a list of the names of any variables, arrays or data structures being passed as parameters.

RUN is used to execute a procedure from the terminal while in the system mode or from within another procedure during program execution.

EXAMPLES:

```
PROCEDURE rundem
(* Show use of RUN statement *)
DIM count:INTEGER
PRINT
FOR count:=1 TO 3
RUN printname
NEXT count
PRINT
END
PROCEDURE printname
(* Procedure to be exercised by rundem *)
PRINT "Dale L. Puckett"
END
```

See also: **PACK**

SAVE

SYSTEM MODE COMMAND

**SYNTAX: SAVE [<proc name> {,<procname>} [>
<pathlist>]]
SAVE* [<pathlist>]**

SAVE writes or "saves" the named procedure — or all your procedures — to a file or device.

SAVE has two arguments; your procedure's name and a pathlist.

SAVE is used to "save" or list your procedures.

See also: **LIST**

INPUT/OUTPUT STATEMENT

SYNTAX: SEEK #<int expr>,<real expr>

SEEK sets the location of a pointer into your disk file. The pointer tells the OS-9 operating system which character to get next if it is **READ**ing or where to put the next character if it is writing.

SEEK needs two arguments. The first is an **INTEGER** expression which represents a pathlist. The second is an expression that evaluates as a real number and is the desired location in the file.

SEEK is used to gain random access to information in your files. It is used with the **GET** and **PUT** statements.

EXAMPLE:

```
PROCEDURE seekdem
(* Show use of SEEK statement *)
DIM path:INTEGER
DIM filepointer:REAL
DIM char:STRING[1]
DIM wholething:STRING[36]
(* First, create a file *)
CREATE #path,"demofile":WRITE
PRINT #path,"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
CLOSE #path
PRINT
PRINT "Here's the entire file we created."

OPEN #path,"demofile":READ
GET #path,wholething
PRINT
PRINT wholething
PRINT
PRINT "Now let's print some specific characters."
PRINT
SEEK #path,10
GET #path,char
PRINT "Character Number 10: "; char
SEEK #path,35
GET #path,char
PRINT "Character Number 35: "; char
SEEK #path,0
GET #path,char
PRINT "Character Number 0: "; char
PRINT
CLOSE #path
SHELL "del demofile"
```

See also: **GET, PUT**

MATH FUNCTION

SYNTAX: SGN(<num>)

SGN returns a -1 if <**num**> is less than zero, 0 if it is equal to zero and 1 if it is greater than zero.

SGN needs one parameter — a number or expression of any type.

SGN is used to tell if a number is positive, negative or equal to zero.

EXAMPLE:

```
PROCEDURE sgndem
(* Demonstrate SGN function *)

DIM sign,number(10):INTEGER

DATA 10,-20,35,45,-56
DATA 0,53,0,-75,89

PRINT

FOR count:=1 TO 10
READ number(count)
NEXT count

PRINT
PRINT "COUNT","NUMBER","SIGN"

FOR count:=1 TO 10
PRINT count,number(count),SGN(number(count))
NEXT count

PRINT
```

See also: none

CONTROL STATEMENT

SYNTAX: SHELL <str expr>

SHELL does not return a value.

SHELL needs one argument — a string expression that represents a command line that will be sent to the OS-9 Operating System.

SHELL is used within BASIC09 procedures to execute OS-9 utilities or programs.

EXAMPLE:

```
PROCEDURE shelldem
(* Show Shell Statement executing OS-9 utilities *)

PRINT "Here are the files in the current data directory."

SHELL "dir"

PRINT

PRINT "And, here are the files in the current execution directory."

PRINT
SHELL "dir x"

PRINT
PRINT "Finally, here is the current date and time: ";
SHELL "date,t"
PRINT \ PRINT
```

See also: none

SIN

MATH FUNCTION

SYNTAX: SIN(<num>)

SIN returns a real number which is the sine of <**num**>.

SIN needs one parameter — a **REAL** number. This number may be expressed in either degrees or radians, depending on the setting of the **DEG/RAD** toggle.

The **SIN**e of an angle is the ratio of the length of the opposite side of a right triangle to the length of the hypotenuse.

See also: **DEG**, **RAD**

SIZE

MISCELLANEOUS FUNCTION

SYNTAX: SIZE(<name>)

SIZE returns the storage size in bytes needed by any variable, array or structure.

SIZE needs one parameter — the name of a variable, array or structure.

SIZE is used with the **SEEK** statement to position your file pointer when working with random access files.

EXAMPLE:

```
PROCEDURE sizedem
(* Show SIZE function *)
DIM example:STRING
DIM exampletwo:STRING[64]
DIM count:BYTE
DIM pointer:INTEGER
DIM number:REAL
PRINT
PRINT "The SIZE function returns the size of any data element."
PRINT
PRINT "Example requires "; SIZE(example); " bytes."
PRINT "Exampletwo requires "; SIZE(exampletwo); " bytes."
PRINT "Count requires "; SIZE(count); " bytes."
PRINT "Pointer requires "; SIZE(pointer); " bytes."
PRINT "And, number requires "; SIZE(number); " bytes."
PRINT
```

```
PRINT "You'll use this function a lot to position the file"
PRINT "pointer in a random file."
PRINT
```

See also: **SEEK**

SQ

MATH FUNCTION

SYNTAX: SQ(<num>)

SQ returns the value of <**num**> squared.

SQ needs one parameter, a **REAL** or **INTEGER** number or expression.

SQ is used to test for square numbers.

See also: **SQR**, **SQRT**

SQRT

MATH FUNCTION

SYNTAX: SQRT(<num>)

SQRT returns the square root of a number or expression.

SQRT needs one parameter — a **REAL** or **INTEGER** number or expression.

SQRT is used to find the square root of a number. It is the same as **SQR**.

EXAMPLE:

```
PROCEDURE sqrtdem
(* Show SQRT function in action *)
DIM number,squareroot:REAL
PRINT
INPUT "Type a number: ",number
squareroot:=SQRT(number)
PRINT
PRINT "The square root of "; number;
PRINT " is "; squareroot
PRINT
```

See also: **SQ, SQR**

SQR

MATH FUNCTION

SYNTAX: SQR(<num>)

SQR returns a **REAL** number — the square root of the number you specify.

SQR needs one parameter — a **REAL** or **INTEGER** number or expression.

SQR is used to find square roots of numbers.

EXAMPLE:

```
PROCEDURE sqrdem
(* Show SQR function in action *)
DIM number,squareroot:REAL

PRINT
INPUT "Type a number: ",number

squareroot:=SQRT(number)

PRINT
PRINT "The square root of "; number;
PRINT " is "; squareroot
PRINT
```

See also: **SQRT, SQR**

STATE

DEBUG MODE COMMAND

SYNTAX: STATE

STATE returns the "calling" order of all active procedures in memory. It is used from the **DEBUG** mode.

STATE does not require any arguments.

STATE is used to find the nesting level of procedures during program execution.

See also: **BREAK**, **PAUSE**

DEBUG MODE COMMAND

SYNTAX: STEP [<number>]
<CR>

STEP executes a suspended procedure a line at a time — or <number> lines at a time. It is entered from the **DEBUG** mode.

STEP needs one argument; a number representing the number of source lines you want to execute at a time. You may enter a carriage return to step through a procedure one line at a time.

STEP is used to watch the flow of a procedure during debugging.

See also: **TRON**, **TROFF**

CONTROL STATEMENT

SYNTAX: STOP [<output list>]

STOP halts your program and returns BASIC09 to the Command Mode.

STOP has one optional argument — an output list that contains a message that may be sent to your terminal.

STOP is used to debug procedures and control program flow.

EXAMPLE:

```
PROCEDURE stopdem
(* Show STOP statement *)

PRINT
PRINT "The STOP statement causes your program to return"
PRINT "to BASIC09's command mode. It can also send a message"
PRINT "to the terminal."
PRINT
STOP "You better give up!  Type RUN to try again!"

(* This line will never be executed *)
```

See also: **END, ERROR**

STR$

STRING FUNCTION

SYNTAX: STR$(<expr>)

STR$ does not return a value. It converts a numeric expression to a **STRING.**

STR$ needs one parameter — a **REAL** or **INTEGER** expression.

STR$ is used to convert a number to a string. The **STRING** may then be manipulated by **STRING** modifiers such as **LEFT$** and **RIGHT$**, etc.

EXAMPLE:

```
PROCEDURE strdem
(* Show STR$ function in action *)
DIM number:REAL
DIM numberholder:STRING
PRINT
INPUT "Type a number at least three digits long: ",number
numberholder: = STR$(number)
PRINT
PRINT "Your number printed as a number: "; " number
PRINT "Your number printed as a string: "; numberholder
PRINT
PRINT "Now, we'll prove it is a string and use the MID$ function"
PRINT "to print the second digit of your number: ";
PRINT MID$(numberholder,2,1)
PRINT
```

See also: **LEFT$, RIGHT$, MID$, ASC**

SUBSTR

STRING FUNCTION

SYNTAX: SUBSTR(<str1>,<str2>)

SUBSTR returns the starting position of <**str1**> in <**str2**> or zero if <**str**> is not found.

SUBSTR needs two parameters — a string to be found and a string to be searched.

SUBSTR is used to find a series of characters within a longer string.

EXAMPLES:

```
PROCEDURE substrdem
(* Demonstrate SUBSTR function *)
DIM example:STRING[50]
DIM pointer:INTEGER

example:= "Now is the time for all good men to sing."

PRINT
PRINT "Here is a string: "; example
PRINT "Let's find the position of the letter 's' in 'sing'."
PRINT

pointer:=SUBSTR("sing",example)

PRINT "The 's' is the "; pointer; "the character."
PRINT
```

See also: **CHR$, LEFT$, MID$, RIGHT$**

TAB

INPUT/OUTPUT STATEMENT

SYNTAX: PRINT [#<int expr>,] [USING <str expr>,] <output list> TAB <expr>

TAB causes BASIC09's **PRINT** statement to output the proper number of spaces to place the cursor at the requested position on a line.

TAB needs one parameter — an **INTEGER** expression that evaluates to the number of spaces you need to **PRINT.**

TAB is used to format your output.

EXAMPLE:

```
PROCEDURE tabdem
(* Show TAB function in use *)

PRINT
PRINT "Let's print some numbers 12 columns apart."
PRINT

FOR count:=0 TO 72 STEP 12
PRINT TAB(count); count;
NEXT count

PRINT
```

See also: **PRINT, PRINT USING**

TAN

MATH FUNCTION

SYNTAX: TAN (<num>)

TAN returns a **REAL** number that represents the **TAN**gent of the angle specified.

TAN needs one parameter — a **REAL** or **INTEGER** number expressed in either degrees or radians. If using degrees, you must toggle the **DEG/RAD** switch with **DEG**. The reverse holds true if you are using **RAD**ians.

TAN is used to find the length of an unknown side of a triangle when you know one side and the angle.

EXAMPLE:

```
PROCEDURE tandem
(* Demonstrate the TAN function *)

DIM angle:REAL

DEG

PRINT
INPUT "Enter an angle expressed in degrees: ",angle

PRINT "The tangent of a "; angle; " degree angle is ";
PRINT TAN(angle)
PRINT
```

See also: **ATN, DEG, RAD**

TRIM$

STRING FUNCTION

SYNTAX: TRIM$(<str>)

TRIM$ returns a **STRING** with all trailing spaces removed.

TRIM$ needs one parameter — a **STRING** or **STRING** variable.

TRIM$ is used to compact strings before storing them in files.

EXAMPLE:

```
PROCEDURE trimdem
(* Show TRIM$ function in use *)
```

```
DIM answer:STRING[32]
DIM trimmedanswer:STRING[32]

PRINT
PRINT "Type your name.  Just for demonstration purposes add"
INPUT "a few spaces after your name: ",answer

trimmedanswer:=TRIM$(answer)

PRINT
PRINT "You typed "; LEN(answer); " characters."
PRINT "There are "; LEN(trimmedanswer); " characters in your name."
PRINT
```

See also: **none**

DIRECTIVE STATEMENT, DEBUG MODE COMMAND

SYNTAX: TROFF

TROFF does not return a value. It turns off BASIC09's trace mode.

TROFF does not require any arguments.

TROFF is used while debugging.

EXAMPLE:

```
PROCEDURE troffdem
(* Show how TROFF stops the trace of your program *)
DIM count:INTEGER

PRINT
PRINT "Let's turn the trace mode on and watch a program."
PRINT

TRON

FOR count:=1 TO 2
PRINT count
NEXT count

TROFF

PRINT
PRINT "The trace mode is now off."
PRINT
```

See also: **TRON**

TRON

DIRECTIVE STATEMENT, DEBUG MODE COMMAND

SYNTAX: TRON

TRON does not return a value. It turns BASIC09's "trace" feature on. After you type **TRON**, each statement of your procedure is decompiled and printed during execution. The value of any variables or expressions evaluated during execution is also printed.

TRON does not require any arguments.

TRON is used to debug your procedures.

EXAMPLE:

```
PROCEDURE trondem
(* Show how TRON traces your program *)
DIM count:INTEGER

PRINT
PRINT "Let's turn the trace mode on and watch a program."
PRINT

TRON

FOR count: = 1 TO 3
PRINT count
NEXT count

TROFF
```

See also: **TROFF**

TRUE

BOOLEAN FUNCTION

SYNTAX: TRUE

TRUE returns the **BOOLEAN** value **TRUE**, which is one of two values that may be returned by a **BOOLEAN** expression.

TRUE does not require any parameters.

TRUE is the result of a **BOOLEAN** operation that makes a logical comparison to test for a certain condition.

See also: **BOOLEAN, FALSE**

DECLARATIVE STATEMENT

SYNTAX: TYPE <typename> :=
<type decl> {; <type decl>}
<type decl> :=
<field name> . <decl> [: type>]
<decl> :=
<name> [<subscript>]

<subscript> :=
(<const> [,<const>] ,<const>]])

<type> :=
BYTE | INTEGER | REAL | BOOLEAN |
STRING [<MAX LEN>] | <user defined>

<user defined> :=
user defined by TYPE statement

TYPE is used to define new BASIC09 data types.

TYPE may use any number of arguments. The complexity depends upon the data type being defined.

TYPE is used to create user defined data structures. After a structure has been typed, storage is reserved with the **DIM** statement.

EXAMPLE:

```
PROCEDURE typedem
(* Show how to enter TYPE statement *)

TYPE labels=name:STRING[20]; address(2):STRING[24]; zip:REAL

DIM mailing_list(50):labels

PRINT
PRINT "Your mailing list will require "; SIZE(mailing_list); " bytes."
PRINT
```

See also: **BOOLEAN, BYTE, DIM, INTEGER, REAL, STRING**

INPUT/OUTPUT STATEMENT

SYNTAX: PRINT [#<expr>] USING <str expr>, <output list>

PRINT USING does not return a value. It is an optional addition to the **PRINT** statement.

PRINT USING needs several parameters. The first is a string expression that determines the format. The second is the output list that you are printing. **PRINT USING** uses the following abbreviations.

R for real format
E for exponential format
I for integer format
H for hexadecimal format
S for string format and
B for Boolean format

PRINT USING lets you format your output so that it is easier to read. There are three additional operators that may be used with **PRINT USING:**

W for field width,
F for fraction field and
J for justification.

EXAMPLE:

```
PROCEDURE printusingdem
(* Show a few examples of PRINT USING *)

PRINT
PRINT "We'll PRINT several examples here.  All will use a"
PRINT "column width of 20 characters."
PRINT
PRINT
PRINT "First, we'll PRINT a few REAL numbers."
PRINT
PRINT USING "R20.2< ",1234.1234
PRINT USING "R20.2> ",1234.1234
PRINT \ PRINT
PRINT "Now let's PRINT a few numbers in exponential format."
PRINT
PRINT USING "E20.2",1234.1234
PRINT USING "E20.2> ",1234.1234
PRINT \ PRINT
PRINT "Let's switch to the INTEGER format."
```

```
PRINT
PRINT USING "I20<",1234
PRINT USING "I20>",1234
PRINT USING "I20↑",1234
PRINT \ PRINT
PRINT "Here are a few Hexadecimal numbers."
PRINT
PRINT USING "H20>",1234
PRINT USING "H20<",1234
PRINT USING "H20↑",1234
PRINT \ PRINT
PRINT "STRING variables may also be formatted."
PRINT
PRINT USING "S20<","Hello Dale"
PRINT USING "S20>","Hello Dale"
PRINT USING "S20↑","Hello Dale"
PRINT \ PRINT
PRINT "And finally, BOOLEAN values may also be sent."
PRINT
PRINT USING "B20<",TRUE
PRINT USING "B20>",FALSE
PRINT USING "B20↑",TRUE
PRINT
```

See also: **PRINT, TAB**

MISCELLANEOUS STATEMENT

SYNTAX: VAL(<str>)

VAL returns a real value.

VAL needs one parameter — a **STRING** variable.

VAL is used to convert a number written as a **STRING** to a number of type **REAL**. It has the effect of stripping off the quotes or dollar sign.

EXAMPLE:

```
PROCEDURE valdem
(* Show use of VAL function *)
DIM stringvalue(3):STRING[12]
DIM numericvalue(3):REAL
DIM count:INTEGER

stringvalue(1):= " 985.34"
stringvalue(2):= "-5796.32"
stringvalue(3):= " -1.2345"

FOR count:=1 TO 3
numericvalue(count):=VAL(stringvalue(count))
NEXT count

PRINT

FOR count:=1 TO 3
PRINT numericvalue(count)
NEXT count

PRINT
PRINT "The STRING values have been successfully "
PRINT "converted to numeric values and printed."
PRINT
```

See also: **LEN, SUBSTR**

CONTROL STATEMENT

SYNTAX: WHILE <bool expr> DO
ENDWHILE

WHILE is part of the **WHILE** ... **DO** ... **ENDWHILE** control structure.

WHILE needs one parameter — a **BOOLEAN** expression.

WHILE is the starting point of the **WHILE** ... **DO** ... **ENDWHILE** construct. A **BOOLEAN** expression is evaluated at the top of this loop. If it evaluates as **TRUE**, the statements inside the loop are executed; when it becomes **FALSE**, the statement following **ENDWHILE** is executed.

EXAMPLE:

```
PROCEDURE whiledem
(* demonstrate use of WHILE ... DO construct. *)

DIM number:INTEGER

number:=0

WHILE number<5 DO
number:=number+1
PRINT "This is loop #"; number
ENDWHILE

PRINT "All done!"
END
```

See also: **DO, ENDWHILE**

INPUT/OUTPUT STATEMENT

SYNTAX: WRITE #<int expr>,<output>

WRITE transfers data to a file or device in ASCII format.

WRITE needs two parameters — The number of a data path and the data to be written.

WRITE stores data in a file or sends it to your terminal or printer. Before you can **WRITE** to a file or device you must **CREATE** or **OPEN** a path.

EXAMPLE:

```
PROCEDURE writedem
(* Show use of WRITE statement *)
DIM words(4):STRING[18]
DIM count,numbers(4):INTEGER
DATA 111,112,112,114
DATA "This","is","a","test."
FOR count:=1 TO 4
READ numbers(count)
NEXT count
FOR count:=1 TO 4
READ words(count)
NEXT count
PRINT
PRINT "We'll WRITE both words and numbers."
PRINT "This time we are using a FOR ... NEXT loop."
PRINT
FOR count:=1 TO 4
WRITE #1,numbers(count)
NEXT count
PRINT
FOR count:=1 TO 4
WRITE #1,words(count)
NEXT count
PRINT
PRINT "Now we are WRITING them from an output list."
PRINT
WRITE #1,words(1),words(2),words(3),words(4)
WRITE #1,numbers(1),numbers(2),numbers(3),numbers(4)
PRINT
PRINT "Notice how the ASCII zeros ($00) that the WRITE statement"
PRINT "sends out do not show up on your terminal."
PRINT
```

```
PRINT "If you had sent the two output lists above to a disk file"
PRINT "you would be able to DUMP the file and see the zeros between"
PRINT "each item in the output list.  A <RETURN> follows the end of"
PRINT "each output list."
PRINT
```

See also: **CREATE, OPEN, UPDATE**

XOR

BOOLEAN FUNCTION

SYNTAX: XOR

XOR returns a **BOOLEAN** value.

XOR does not require any parameters.

XOR is used in an **IF** ... **THEN** statement as an eXclusive **OR** operator. For example:

```
IF firstnumber = 5 XOR secondnumber = 5 THEN 100
```

This statement reads, "If firstnumber is equal to 5 **OR** second number is equal to 5, but not both of them, **THEN** go execute line 100."

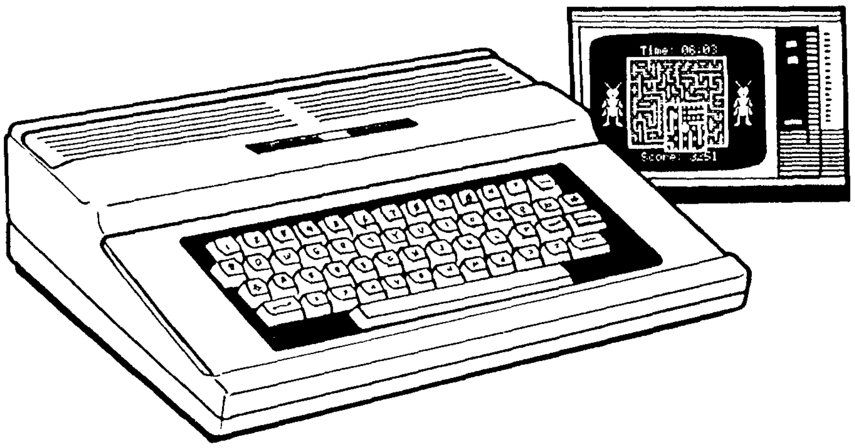
EXAMPLE:

```
PROCEDURE xordem
(* Show XOR operator in use *)
DIM number,numbertwo:INTEGER
DIM status:BOOLEAN
PRINT
INPUT "Type a number between one and 10: ",number
INPUT "Thank you.  Please type one more in the same range: ",numbertwo
(* 'status' becomes true if number=3 OR if numbertwo=8 *)
(* — except when number=3 AND numbertwo=8 *)
(* When both these conditions are TRUE the value of 'status' *)
(* becomes false. *)
status:=number=3 XOR numbertwo=8
IF status THEN
PRINT
PRINT "One condition or the other is TRUE.  Congratulations."
ELSE
PRINT
PRINT "Whoops!  Neither of the conditions tested are TRUE."
PRINT "Or, you somehow managed to guess both numbers correctly."
ENDIF
PRINT
```

See also: **AND, NOT, OR**

PART V

color computer graphics



This is a special section for owners of TRS-80 Color Computers. A Color Computer wouldn't be a Color Computer without graphics, so the Color Computer version of BASIC09 gives you an excellent selection of powerful graphics commands.

One preliminary note: the information of this chapter is based on the features of the Color Computer 1 or Color Computer 2, BASIC09 Version RS 1.00.00, and OS-9 Level One Version RS 1.01.00, unless otherwise noted.

THE COLOR COMPUTER DISPLAY

Before we dig into the nitty-gritty of the graphics commands themselves, we'll need to review how the Color Computer's display system works.

The Color Computer can display a screen of 512 **alphanumeric** characters or a screen of **graphics.** The alphanumeric mode is what you've been using all along to communicate with BASIC09 and OS-9. The graphics mode is what many popular Color Computer entertainment programs use to draw detailed pictures.

BASIC09 gives you a choice of two graphics screen formats. The first is a grid of 24,576 individual points. The grid is organized into 128 columns across the screen and 192 rows up the screen. Each point can be individually set to one of four colors. You can think of this as "color TV" mode.

The second graphics screen format is a grid of 49,152 points organized into 256 columns and 192 rows. Each point can be turned on or off only (i.e., this is a two color mode). This can be compared to "black and white TV". Comparing this format to the four color format, you are trading color for more picture detail.

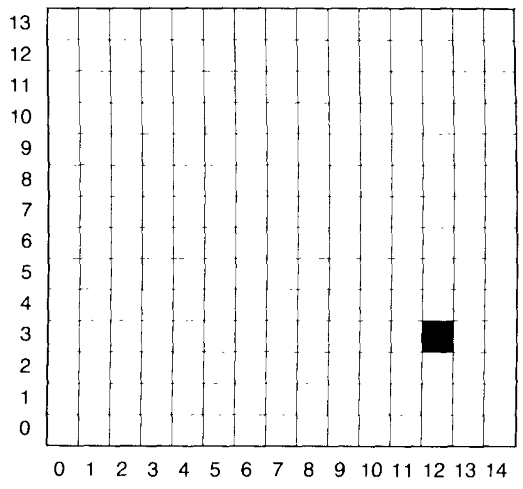
These two graphics modes are the most powerful that are supported by the Color Computer hardware. They roughly correspond to the limit of what a consumer model color TV can display using a connection through its antenna terminals. The Color Computer hardware can also produce less detailed screens but because these are so rarely used OS-9 does not support them.

Unfortunately, mixing characters and graphics images on the same screen is not taken care of by the hardware. It can be done by software, but it's not easy unless you are a fairly skilled programmer. It can also take up a lot of memory. Therefore, BASIC09 gives you commands that lets you switch back and forth between separate graphics and text screens, but not both at the same time.

SCREEN COORDINATES AND THE GRAPHICS CURSOR

The rows and columns of points on the screen are a coordinate system which is used by all graphics drawing commands. Coordinates are a pair of numbers used to refer to particular points such as the middle of a circle or the end point of a line. The rows and columns are numbered from zero upward beginning at the lower left-hand corner of the screen.

The column number is given first. This makes graphics coordinates the same as the common mathematical "X/Y" coordinate system. For example, the coordinate (12,3) refers to a point at the intersection of the thirteenth column from the left and the fourth row up from the bottom.



No, we're not trying to put you on! If this sounds wrong to you, you have already forgotten that numbering begins at line zero, column zero. Count the lines in the example below if you can't see why this causes "coordinate inflation". Numbering things starting with the "zeroth" item is another one of those little traps designed to keep you on your toes.

Many of the graphics commands can use coordinates using the current position of a **graphics cursor.** The graphics cursor is very similar to the cursor used in the alphanumeric code. It indicates the place where the next drawing command will start. It is moved automatically by many commands, or can be moved to a particular point using the **MOVE** command. One big difference compared to the alpha mode cursor is that the graphics cursor is invisible. This keeps it from becoming an unintended part of your pictures.

COLOR CODES AND SETS

Depending on the screen format you select, you can use at most two or four different colors at a time. When you use a graphics drawing command, you specify the color you want according to a color code number.

The color code also has another important use. It controls which one of the several available color sets will be used. This gives you a wider selection of colors to choose from. The color codes and corresponding color sets are given in the table below.

		256 By 192 Two Color Format		128 by 192 Four Color Format	
	Color Code	**Background**	**Foreground**	**Background**	**Foreground**
	00	Black	Black	Green	Green
	01	Black	Green	Green	Yellow
Color	**02**			Green	Blue
Set 1	**03**			Green	Red
	04	Black	Black	Buff	Buff
Color	**05**	Black	Buff	Buff	Cyan
Set 2	**06**			Buff	Magenta
	07			Buff	Orange
	08	not used		Black	Black
Color	**09**			Black	Dark Green
Set 3	**10**			Black	Med. Green
	11			Black	Light Green
	12	not used		Black	Black
Color	**13**			Black	Blue
Set 4	**14**			Black	Red
	15			Black	Buff

Notice that the two-color mode only has two color sets—black/green and black/buff ("buff" is very close to white). The four color gives you four choices. Color sets 1, 2, and 4 offer useful combinations of different colors. Colors set 3 is interesting because it is monochrome (black/green) with different intensities, which offers shading and contouring possibilities.

Color set selection is automatic when you give any color code. For example, if you are in four color mode and you draw a line using color code 07, color set 2 is automatically selected. You can change color sets as you wish. The only effect on the screen is that the colors change to the corresponding colors of the new set. Continuing the example, if you next draw a line using color set 4, the orange line (color code 5) drawn previously will change to buff, which is the coresponding color in set 4.

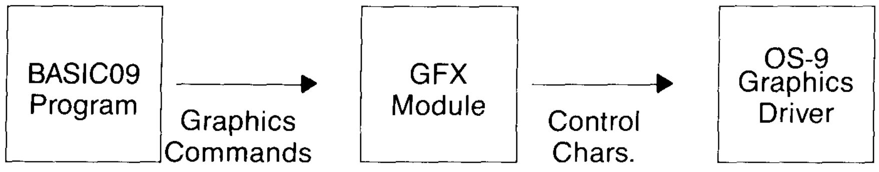
Note that the background color varies according to the color set selected. The background color is also always one of the two or four colors available. To erase something, you simply redraw it in the background color.

You can set the "foreground color" to any color. The foreground color is the default color the drawing modes will use if you don't specifically state your choice in each command. This is especially convenient when you use a sequence of drawing commands that will use the same color. It saves typing and program size.

THE GFX MODULE

Technically, BASIC09 does not have built-in graphics commands. Most of the actual drawing is done by OS-9's graphics driver module. Because of this, the graphics functions can be used by any programming language. Basic graphics commands are passed to OS-9 via control character sequences. Overall, this is a very flexible system but control character sequences are never a lot of fun to work with.

The Color Computer version of BASIC09 comes with a special module called "GFX" which makes graphics programming quite simple. It effectively adds a whole new set of commands to BASIC09 using the **RUN** statement.



GFX can be thought of as a "black box" which takes high-level graphics commands and translates them to the corresponding control character sequences. GFX is so automatic you don't even need to know anything about what the control characters are or how they work. The illustration below shows the internal flow of graphics commands from your program to OS-9 through GFX.

The file containing the GFX module should be kept in your commands directory. It will be automatically loaded whenever you use it. You can also manually pre-load it into memory using the following OS-9 command line:

load gfx

Running a GFX graphics command is quite simple. The **RUN** statement is used. The name of the command is passed as parameters, followed by additional parameters as necessary. Here are some examples:

```
RUN gfx("clear")
RUN gfx("line",20,20,2)
RUN gfx("color",7)
```

Unless otherwise noted, the coordinate and color parameters are always of type INTEGER.

A SUMMARY OF GFX FUNCTIONS

HOUSEKEEPING FUNCTIONS

ALPHA –	**switches back to the alphanumeric screen**
COLOR –	**sets the current drawing foreground color**
MODE –	**selects a graphic screen and a screen format**
MOVE –	**changes the current position of the graphics cursor**
QUIT –	**exits the graphics mode, giving back screen memory**

DRAWING FUNCTIONS

CIRCLE –	**draws a circle**
CLEAR –	**erases or presets the entire screen**
LINE –	**draws a line**
POINT –	**sets the color of a specific point**

STATUS CHECKING FUNCTIONS

GLOC –	**returns the current position of the graphics cursor**
GCOLR –	**returns the color of the point under the graphics cursor**
JOYSTK –	**returns the current position of the joystick(s) or mouse**

ENTERING THE GRAPHICS MODE

The **MODE** command switches the screen from the normal alphanumeric mode to either of the two graphics modes. You must state the screen format and initial foreground color you desire. **MODE** also clears the screen the first time it is used. The format of the **MODE** command is:

```
RUN gfx("mode",format,color)
```

The 'format" code must be 0 for two color mode or 1 for four color mode. The "color" code selects the initial foreground color. Remember that this also selects the associated color set. You must execute a **MODE** command before you can execute other graphics commands.

The first time the **MODE** command is used it will ask OS-9 for a 6K byte memory space for screen display memory. If there isn't 6K of free memory available in one chunk, you'll get a "memory full" error.

This will also cause GFX to be automatically loaded if you didn't preload it some time before. GFX will also need a bit of memory to load in (about ½K). Therefore, it can also cause a memory full error.

Memory full errors usually happen when you gobble up almost all of the system's free memory when you start up BASIC09. The cure is to exit BASIC09, and reenter it, this time asking for about 7K or 8K less workspace memory.

Another related point is that GFX will keep the 6K of screen memory until you use the **QUIT** command to permanently exit the graphics mode.

THE HOUSEKEEPING COMMANDS

These commands generally control the graphics environment.

The **ALPHA** command temporarily switches the screen back to the alphanumeric mode. The graphics and alphanumeric modes each have a separate display memory area. This lets you switch back and forth without affecting what is displayed on either screen. A subsequent **MODE** command will switch back to the graphics screen. The **ALPHA** command never has parameters, just:

```
RUN gfx("alpha")
```

The **COLOR** command lets you change the current foreground color at any time. Remember that your choice of color code also selects the current color set. For example, to set the foreground color to magenta (color set 2 in four color mode) you would use:

```
RUN gfx("color",6)
```

The **MOVE** command lets you change the location of the invisible graphics cursor. You must give the desired coordinates, for example:

```
RUN gfx("move",63,127)
```

The **QUIT** command is used when you are completely done using the graphics modes, because it gives back the 6K of display memory. It's a good idea to always do this at the end of your program or you may "lose" 6K of useful memory. This command is extremely simple:

```
RUN gfx("quit")
```

THE DRAWING COMMANDS

This is the fun part. These commands create pictures.

Most of the drawing commands share three conveniences features. First, the color code (always the last parameter) is optional. If you don't give one, the current foreground color will be used. Second, if the command requires one or two coordinates, the first set is optional. If you omit one coordinate, the current graphics cursor location will be used in its place. And finally, many of the commands automatically update the cursor position.

The function of **CLEAR** is simple: it clears the screen. It also resets the graphics cursor location to 0,0. It has another use: if you specify a color it will preset the entire screen to that color. Here are some examples:

```
RUN gfx("clear")
RUN gfx("clear",1)
```

POINT sets the color of a single point on the display to a specified color. It also sets the graphics cursor to that point. Remember that if you omit the color code, the current foreground color will be used.

```
RUN gfx("point", NewX, NewY)
RUN gfx("point",50,99,5)
```

LINE draws straight lines in any direction. You can give the **LINE** command two coordinates for the end points of the line. Alternatively, you can give just one coordinate. In this case the position of the graphics cursor is used at the starting point, and the coordinate you give is used as the end point. The graphics cursor position is always moved to the end point of the line. This lets you easily draw a series of connected lines by just giving the end point of each line.

Like the other drawing commands, you can either give a specific color code, or the current foreground color will be used automatically.

The next example draws a border around the screen, taking advantage of the foreground color and graphics cursor features to keep the commands as brief as possible.

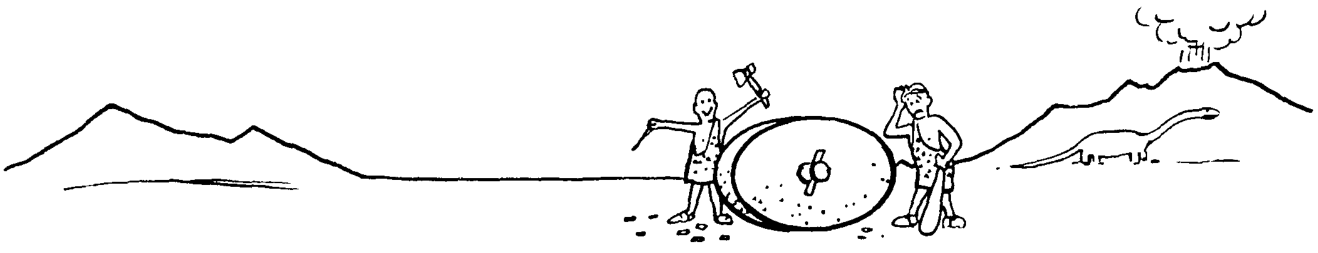
```
RUN gfx ("mode",1,2)
RUN gfx("color",4)
RUN gfx("line",0,0,0,191)
RUN gfx("line",255,191)
RUN gfx("line",255,0)
RUN gfx("line",0,0)
```

Another useful command is **CIRCLE.** This command needs to know where you want the center of the circle and the desired radius. If you don't give coordinates for the center point, the current position of the graphics cursor will be used. You must always give the radius. The color code, once again, is optional. Here's an example that draws a circle centered in the middle of a 4-color mode screen, with a radius of 64 and a color code of 7:

```
RUN gfx("circle",128,96,64,7)
```

If you let the system use the current foreground color and the current graphics pointer position, you get the simplest form of the command where only the radius is given:

```
RUN gfx("circle",64)
```



Circles may not look perfectly round to you. If so, don't run out to get your eyes checked or your computer fixed. Circles may look slightly oval-shaped because GFX draws them to be mathematically correct with respect to the point grid. The catch here is that the grid isn't perfectly square because your TV screen isn't! This causes things to have a slight vertical stretch. You may have also noticed this in certain film broadcasts.

Round but mathematically incorrect circles might look a little better but they would drive you crazy trying to make them connect to other things.

GFX has three commands that return useful information.

JOYSTK lets you read the current status of either of the joystick ports. There must be exactly four parameters.

The first is an **INTEGER** in which you give a code number to select the right (0) or left (1) port. If you are using both ports you have to read them individually.

The second is a **BYTE, INTEGER,** or **BOOLEAN** (your choice) value which is used to return the status of the fire button. A non-zero (or **BOOLEAN TRUE**) returned value indicates that the button is being pressed.

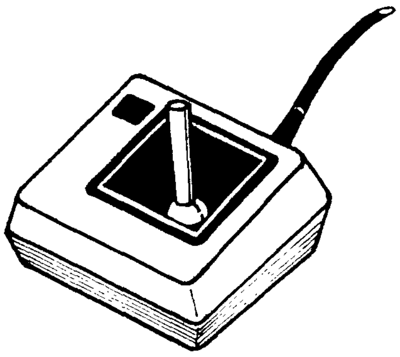
The third and fourth values may be of type **INTEGER** or **BYTE** and are used to return the current X, Y position of the joystick or mouse.

The value returned will range from 0 to 63. Here is an example:

```
DIM JoySelect,JoyFire,JoyX,JoyY:INTEGER
JoySelect := 0
RUN gfx("joystk",JoySelect,JoyFire,JoyX,JoyY)
```

If you want to convert raw joystick readings to a full-sized screen coordinate, you must multiply by the factors given in the table below.

mode	X	Y
2 color	4	3
4 color	2	3



GCOLR is used to read the color code of a specific point on the screen. You can specify the coordinates of the point, or let the system automatically use the current graphics cursor position. You must provide a **INTEGER** or **BYTE** type variable to hold the returned color code value. For example:

```
DIM WhatsUnder:BYTE
RUN gfx("gcolr",25,40,WhatsUnder)
```

The last status command, **GLOC,** is intended for graphics wizards. It returns the physical address of the video display **RAM.** This can be used with **PEEK** and **POKE** statements to directly diddle bits in the display **RAM. POKE** is especially dangerous if you don't know exactly what you're doing, so we'll leave this one alone.

WHERE'S THE PAINT COMMAND?

PAINT is a very handy command. It fills in an enclosed area of any shape with your choice of color.

The catch is that GFX does not have a built-in **PAINT** command, but you're OS-9 system might. If you have version OS-9 version RS 02.00.00 or later, there is a paint function in the graphics driver. However, because GFX was designed before the time 02.00.00 was released it has no **PAINT** command.

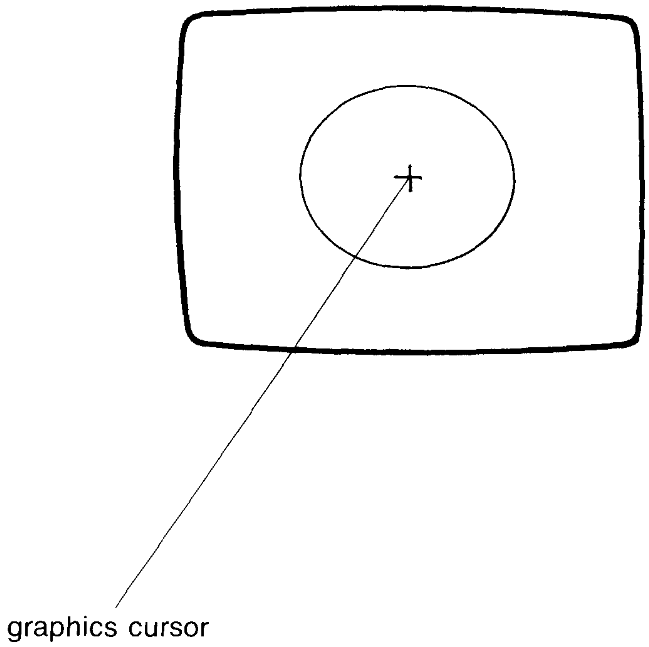
But don't panic—if you do have the most current version of OS-9, using the **PAINT** command using a control code is very simple. Just do this:

```
PRINT chr$(29);
```

This will fill the area enclosing the current graphics cursor position with whatever the current foreground color is. The painting stops whenever it hits a point that is a different color than the original color of the point under the cursor.

Here is an example of how to draw a circle and fill it.

```
RUN gfx("clear")
RUN gfx("move",128,96)
RUN gfx("color",1)
RUN gfx("circle",64)
PRINT chr$(29);
```



Index To Sections I and II

notes

microware®

MICROWARE SYSTEMS CORPORATION

1866 NW 114th Street
Des Moines, Iowa 50322

ISBN 0-918035-00-7