ATARI 400/800™

BASIC REFERENCE MANUAL

February, 1980

PRELIMINARY

PREFACE

This is not a novel, nor have we attempted to make it read like one. It's a **Reference Manual** for beginners in personal computing, and it covers the complex subject of computer programming in **ATARI® BASIC**. A computer is unique in the world of machines. It has *no* specific design function, but rather is intended to do what its user tells it to! Therefore the new programmer must learn a language that both *he* and the *computer* can understand!

This language is called **BASIC**, which is short for **B**eginners **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode. This **BASIC** computer language is what this book is all about. **ATARI BASIC** differs somewhat from "traditional" or "standard" BASIC for the purpose of accentuating the particular capabilities of the ATARI Personal Computers.

Just like applications for a home computer, users come in all varieties! They also come at all levels of skill and experience and with all levels of need for information. If you have had previous programming experience in **BASIC**, you will find many features of **ATARI BASIC** that are old friends. There are also some new ones that are used only with ATARI Personal Computer Systems.

This **BASIC Reference Manual** is our attempt to cover all the material most owners could want, to cover it at all levels of user experience and skill, and simultaneously to try to put all this information into a single easy-to-read, "handy" format. Big job! Hope we succeeded!

Hope you enjoy using your ATARI Personal Computer, and have fun with our little tour through what we call "BASICland". Good luck and happy computing!

CHAPTER 1

INTRODUCTION

1.1 General Information

A thorough study of this chapter and Appendix F (covering use of the keyboard) is strongly recommended. They will tell you about the organization of the Manual and how to find what you need. This chapter will give you the "keys" that will unlock some of the esoteric terminology of the computer world, (see also Appendix G, the Glossary) and will point up the major features of **ATARI BASIC** that are unique or unusual. Probably the most deserving of your immediate attention is the section that tells you about the notation conventions used in this book to describe commands, functions, and other BASIC statements. Once you *really understand* these conventions, you will find the rest of the book (and most other computer software reference books) easy to read and understand. Each BASIC term is discussed using a certain format as explained in the notation conventions section of this chapter. There are program examples throughout the book that show you how the statement under discussion might look when actually "keyboarded" into your computer. Just one more time: *Study the section on notation conventions!* A few minutes invested here will save you hours of frustration later.

1.2 Features of ATARI BASIC

Some of the special features of **ATARI BASIC** include: Sound effects with up to four independently controlled voices, five octave range, and programmer control of tonal qualities and volume; three text modes and five graphics modes; full, 16 color graphics control; string length limited only by memory capacity, similarly "unlimited" loop nesting capability; special game controller access using BASIC commands (not **PEEK** or **POKE**); floating point functions to nine digits; device independent input/output operations; and readily usable **PEEK, POKE,** and **USR** capabilities for those of you who want to really get in there and meet your computer at its own machine language level! Variable names limited only by line length are also unusual features, as are the "fingertip programmable" special keyboard graphics characters. See Appendix F for information on the keyboard and the editing system. The latter is in itself very different and very powerful!

1.3 Terminology: Stranger in a Strange Land?

Books on BASIC (and computers in general) are full of complex sounding terminology that the beginner must either master at once or shake his head in wonder until he does. The former is much to be preferred...besides, think of how you can impress your friends

who haven't bought their ATARI Personal Computers yet! Seriously though, we will make every attempt to keep the "jargon" to a minimum, and to explain our terms as we go. Hopefully, a quick side-tour now and then to the Glossary/Index in Appendix G will be necessary only occasionally, and mostly for reference purposes (unless you just like side tours). Let's start with a few terms, presented as painlessly as possible:

BASIC Statement: Analogous to a sentence in English. A statement has a *verb* (such as **LET, PRINT, RUN,** =, >=, etc.,) a *subject* (very often a *variable* or a *constant*), and sometimes an *object* (usually an *expression*) All parts of a statement must, of course, be "legal" in BASIC... that is, the computer must be able to recognize and interpret them (hopefully as desired by the programmer!) Sometimes parts of a statement may be omitted or "implied" just as in an English sentence. For example, we may say "Let us set the value of X equal to one." Or we may say (equally correctly) "Let X=1". Or even "X=1". Many of the same sentence contractions are permissible in BASIC, as we shall see.

Variable: A variable is simply the use of a *name* for a numerical or other quantity which may (or may not!) vary. Think of a variable as a little box in which we can store a value. We can, for example, take the number 100 and store it in the variable named X. We do this by saying **LET X=100**. We can store the 100 in any legal variable. Note here that variable names (such as X in our example) may be up to 120 characters long. However, a variable name *must* start with a capital letter, and may contain *only* capital *letters* and numerical *digits*. No punctuation marks, graphics symbols, etc., please! Here are a few more examples of storing a value in a variable:

```
LET C123DVB=1.234
LET VARIABLE112=267.543
LET A=1
LET F5TH7=6.5
LET THISNO=59.809
```

(Note: **LET** is optional and may be omitted)

Constant: This is just a teeny tiny bit of "computerese": We normally think of a constant as something that never changes (like the value of Pi). In our new electronic context, however, a constant is just a value expressed as a number rather than represented by a variable name! For example, if we tell the computer that **X = 100**, X is a **variable** and 100 is a **constant**.

Expression: This is like a phrase in English it can be evaluated to a numerical quantity. For example, the following is an expression: 4+5 - 6+10. It evaluates to the numerical quantity 13. An expression may be either logical or (as in our example) arithmetic. It may be made up of any legal combination of variables, constants, operators, and functions used together to compute a value. Note that expressions are usually (but not always) found to the right of an = sign as in the typical **LET** statement: X=4+5-6+10.

Function: A function is a computation built into the computer's system so that it can be called for by the user in a program. A function is *NOT* a statement! Either it **is** an expression, is part of one, or is used with an expression. It is really a *subroutine* used to compute the value of the function, which is then "returned" to the main program when the subroutine returns (more on subroutines in Chapter 3). But a function does nothing in and of itself that the user can see. **COS** (cosine), **RND** (random), **FRE** (unused memory space), and **INT** (integer) are examples of functions. Since functions do no more than return a value to the program, we must always tell the computer to do something with the value returned. In many cases the value is simply assigned to a variable (stored in a variable) for later use. In other cases it may be printed out on the screen immediately. See Chapters 6 and 7 for more on functions. Meanwhile, here are a couple of examples of functions as they might appear in programs:

```
10 PRINT RND(0)          (print out the random number returned)
10 X=100+COS(45)         (add the value returned to 100 and
                          store the total in variable X)
```

String: A string is a group of characters "strung" together and enclosed in quotation marks: "ABRACADABRA" is a string. So is "ATARI MAKES HAPPY COMPUTERS". Likewise "123456789" is a string. A string is much like a constant, for it too, may be stored in a variable! A **string variable** is a little different, though, for its name *must* end in the character **$**. For example, the string "ATARI 800" may be assigned to a variable called A$ using (optional) **LET** like this:

```
10 LET A$="ATARI 800"     (note quotation marks)
          OR
10 A$="ATARI 800"         (LET is optional, the quotes are re-
                           quired!)
```

Because the quotation marks are used to tell the computer that a string is being started or stopped (called "delimiting" in computerese), quotation marks may **not** be used within the string itself. See Chapter 7 for a complete dissertation on strings!

Arrays and Array Variables: Think of an **array** as a **list** of places where data can be filed away for future use. Each of these places is called an "element", and the whole array or any element is an array variable. For example, let's consider an array that we will call "array A" as having 6 elements. These elements are referred to by the use of **subscripted** variables such as A(2), A(3), A(4), etc. A number can be stored in each of these elements.

This may be accomplished element by element (using the **LET** statement), or as a part of a **FOR/NEXT** loop (see Chapter 8).

Note: Don't leave any blanks between the element number in parentheses and the name of the array! This will cause your computer all kinds of confusion, which it will throw right back into your face!

Do it like this:	**Never like this:**
A(23)	**A (23)**
ARRAY(3)	**ARRAY (3)**
X123(38)	**X123 (38)**
	These spaces are "no-nos"!

1.4 Notation Conventions Used in this Manual

The following conventions are applied throughout this Manual. If the reader will take a little time before proceeding to **learn the conventions**, he will have **much** less trouble understanding some of the format explanations used throughout the rest of the text.

Line Format: The format of a line in a BASIC program includes a **line number** (abbreviated to lineo) at the beginning of the line, followed by a **statement keyword**, followed by the body of the statement, and ending with a **line terminator** command. The terminator in the case of the ATARI Personal Computers is simply pushing the RETURN key. In an actual program, the four elements might look like this:

Line Number	Keyword	Body	Terminator
10Ø	PRINT	A/X*(Z+4.567)	RETURN

Remember: Every line in BASIC program **must** begin with a line number and **must** end with the terminator command RETURN .

CAPITAL LETTERS: In this book, CAPITAL LETTERS denote keywords which *must* be typed by the user in upper case form *exactly* as they are printed in this text. Reverse-video characters will not work. Here are a few examples:

PRINT RUN LIST END GOTO GOSUB FOR NEXT IF

(See Appendix B for a complete listing of all BASIC keywords.) As a practical matter, we can say that in a program, **all** alphabetical characters used **other than within quotation marks** (i.e., in strings) **must** be capitalized and **must not** be reverse video characters.

Lower Case Letters: In this Manual, lower case letters are used to denote the various classes of items which may be used in a program, such as variables (var), expressions (exp), and the like. The abbreviations used for these classes of items are shown in Table 1.1 below.

Items in Brackets: Square brackets like these [ ] contain *optional* items which **may** be used but are **not required**. These brackets must not be confused with parentheses like these ( ) which are entirely different!

Items in Brackets followed by Three Dots [exp...]: These items are optional, but may be used in any number desired. Thus [, exp, exp, exp] and so on. The dots just mean (in this case) that any number of expressions may be "tacked on" as needed, but **none** are required. Note that the leading comma is used only if the brackets are used. It, too, is optional, but **must** be used between each pair of exps.

Items stacked vertically in braces: Items stacked vertically in braces mean that any one of the stacked items may be used, but that only one at a time is permissible. Here's a good example of this:

lineno { **GOTO** / **GOSUB** } **lineno**

(**Here either GOTO** or **GOSUB** may be used, but not both!)

TABLE 1.1

ABBREVIATIONS USED IN THIS MANUAL

Abbreviations	Explanation and Discussion of Items
avar	**Arithmetic Variable:** A location where a numeric value is stored. Variable names may be from 1 to 120 alphanumeric characters, but must start with an alphabetic character, and all alpha characters must be unreversed, upper case.
svar	**String Variable:** A location where a string of characters may be stored. Same name rules as avar except that the last character in the variable name **must** be a **$**.
mvar	**Matrix Variable:** (Also called a **subscripted** variable). An element of an array or matrix. The variable name for the array or matrix as a whole may be any legal variable name such as A, X, Y, ZIP, K, or whatever. The subscripted variable (name for the particular element) starts with the matrix variable, and then uses a number, variable, or expression in parentheses immediately following (**without** a space!) the array or matrix variable. For example, A(ROW), A(1), A(X+1).
var	**Variable:** Any variable. May be **mvar, avar,** or **svar.**
aexp	**Arithmetic Expression:** Generally composed of **"aexp aop aexp"** where each aexp may be an **lexp, avar, mvar, constant,** or arithmetic function. Any arithmetic expression...
lexp	**Logical Expression:** Generally composed of **aexp lop aexp** or **sexp lop sexp.** Such an expression evaluates to either a 1 (logical true) or a 0 (logical false). For example, the expression 1<2 evaluates to the value 1 (true) while the expression "LEMON" = "ORANGE" evaluates to a zero (false) as the two strings are not equal.
sexp	**String Expression:** Can consist of a **string variable, string literal** (constant), or a **function** that returns a string value. No operators are allowed in **sexp.**
aop	The arithmetic operator signs: +, −, *, /, <, <=, =, >=, <>, ^
lop	The logical operators: **AND, OR, NOT** (the latter is a special case: **NOT** is a unary operator)

lineno	A **constant** that identifies a particular program line in a deferred mode BASIC program.
exp	Any expression, whether **sexp** or **aexp.**
adata	Data expressed in ATASCII code. (See Chapter 5 and Appendix E.) Quotation marks are "thrown out" by the computer, as are leading blanks.
"filespec"	Reference to a particular file, **usually** on disk. **"file spec"** always appears in quotes, and contains information on the type of I/O device, its number, a colon, an optional file name, and an optional filename extender. Example filespec: "D1:NATALIE.ED" (See Chapter 5, especially Figure 5.2)

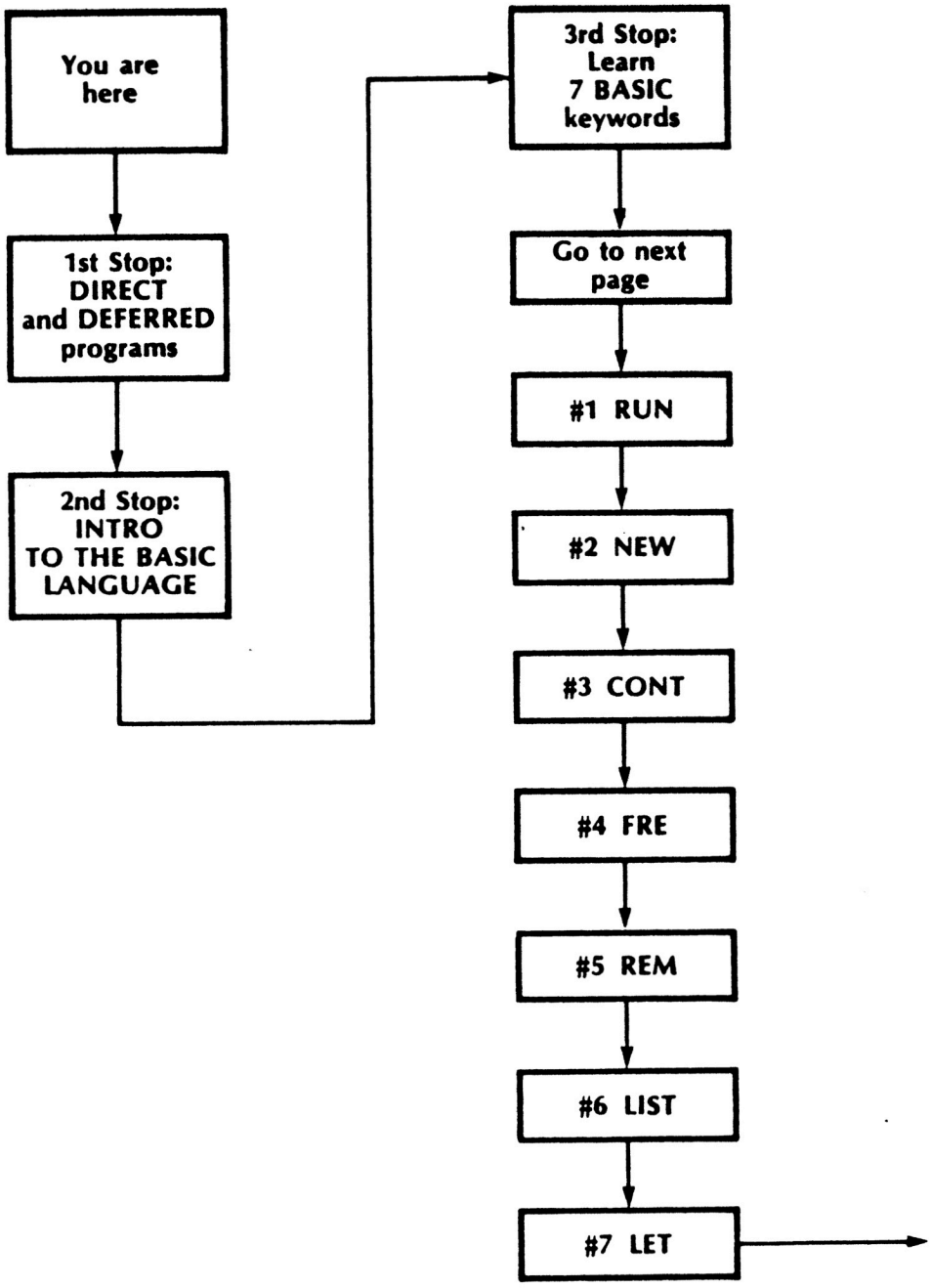
1.5 And Now! ONWARD! Into BASICland!

Welcome to the strange and wonderful place called BASICland! You know enough now to begin a serious study of the language spoken here, for you have been introduced to a few of the words. More important, you know how to use the handy guide-book.

In the rest of this book, you will get a guided tour of the place, and you will meet a few of the natives. Luckily, most of these are friendly, and your relationships will be helped along by the time you have spent studying this introductory material.

— Have a Nice Trip! —

BEGINNINGS IN BASIC: PROGRAM AND SEQUENCE CONTROLS



2.1 Direct and Deferred Programs

Before we get into the mysterious realm of the BASIC computer language, we need to look at the two "modes" that most computers including ATARI Personal Computer Systems using BASIC can employ. In computer lingo a "mode" is a method, type, or class of operation. The term is used like this: Input **mode** (as opposed to output **mode**), graphics **mode** (as opposed to text **mode**), and in the case we are about to examine **Direct Mode** (as opposed to **Deferred Mode**).

Remember in Chapter 1 when we discussed the format of a BASIC line, we said that the first entry on such a line was the line number? True, in **Deferred Mode**, where execution is delayed until the computer is commanded to **RUN**. But there is also a **Direct Mode** (similar to a calculator, but much more powerful) which **RUN**s executable statements **as ooon as the RETURN key is pressed.** In **Direct Mode**, line numbers are not used, and that's how the computer is told to use this mode. The number of statements that can be entered in a single **Direct Mode** program is limited by the length of the logical line. Several statements can be entered, separated by a colon (:). Here is an example of a **Direct Mode** program:

READY	(computer prints)
LET X=1: LET Y=2: PRINT X+Y RETURN	(you type)
3	(computer prints)
READY	(computer prints)

Here's another example:

READY	(computer prints)
PRINT "MY OWNER LOVES ME!" RETURN	(you type)
MY OWNER LOVES ME	(computer prints)
READY	(computer prints)

Note: This would be a good time for you to "power up" your system and practice entering a few simple **Direct Mode** programs. Remember to press **RETURN** when you are ready to **RUN** your program, and remember that it won't show you anything (except perhaps an ERROR message!) unless you tell it to **PRINT** what you want to see on the screen.

You are quite limited in **Direct Mode**, but you will find it very handy for calculator-type operations, and for making calculations during times when you are entering a program in **Deferred Mode**. It is also useful when there is already a program stored in memory that you don't want to disturb to do a calculation. Remember that you can use **Direct Mode** as much as you like without disturbing what is stored in the computer's memory, *provided* you don't use the same variables in both modes at the same time!

The drawbacks of **Direct Mode** are its program length limitations (120 characters maximum), and the fact that each time you wish to **RUN** your program you must retype the whole thing into the machine prior to hitting the RETURN key. The cure for all this is the **Deferred Mode**.

If you use line numbers, the computer will automatically place itself (and you!) in **Deferred Mode**. This means that the program can be as long as you like (limited only by the amount of memory you have purchased), need only be entered *once*, can be stored using the program recorder or diskette and re-run any time you desire, and can be modified for different purposes at anytime without having to retype the whole thing.

Now try a couple of little programs in **Deferred Mode.** If you are a beginner, just type them into the computer without worrying at this point about what the words really mean. We'll cover every BASIC keyword in the rest of this manual. For now, just familiarize yourself with the **Deferred** programming **Mode**.

Examples Of A Simple Program In Deferred Mode

READY		(computer displays)
10 A=3	RETURN	(you type)
20 B=A+2	RETURN	(you type)
30 PRINT "A PLUS B IS "; A+B	RETURN	(you type)
RUN	RETURN	(you type)
8		(computer displays)
READY		(computer displays)

READY		(computer displays)
NEW	RETURN	(you type - **Note no line number)**
10 A=45	RETURN	(you type)
20 X=5		(you type)
30 PRINT "A DIVIDED BY X IS "; A/X	RETURN	(you type - **Note one space after "is" and before closing quote)**
RUN	RETURN	(you type)
A DIVIDED BY X IS 9		(computer displays)
READY		(computer displays)

Notes on the above routines: Consult Appendix F of this manual for information on the keyboard. Try changing the values of A and X in the above program. The words **NEW** (which clears out the old program from memory) and **RUN** (which causes the computer to execute your program), are commands normally performed in **Direct Mode** and require no line numbers. If you placed them in numbered lines, they wouldn't work immediately. You would have to repeat them later in direct mode (no line numbers) anyhow. Remember, through, that any BASIC statement is "legal" in both modes.

2.2 A Few More Introductory Remarks About BASIC

Before we discuss the actual BASIC commands and other keywords, we should briefly review a few items and comment on a couple more: Remember that in **Deferred Mode**, each BASIC line requires a line number, a BASIC statement or keyword, the body of the statement, and the pressing of the RETURN key, in that order. However, we should also be aware that just as in **Direct Mode**, we can enter more than one BASIC program statement on each line in **Deferred Mode**. In this case, too, we must use a colon (:) to separate the statements. For example:

10 X=3: Y=X+2: Z=X+Y:PRINT Z RETURN

Line numbering: Note that the line number used can be any number between 0 and 32767. Fractional line numbers (such as 40.6 or 60.3) are rounded off to the nearest whole integer (not rounded down as with the **INT** function!) Also, no negative (−) line numbers, please...

The line numbers determine the order of program execution. There are several ways in which the sequence can be modified (with **GOTO, GOSUB, TRAP**, etc.), but without such "outside" direction, the computer will execute the program in the numerical order of the line numbers. **This is true regardless of the order in which you type in the lines!** For example, if you use the order 100, 10, 20, 200, and 30 when you enter source program material through the keyboard, the sequence of execution will **still** be 10, 20, 30, 100, 200.

Note in the above example that line numbers need not be consecutive integers; that is, spaces of unused numbers may be left between the numbers actually used. This is good practice as in this way additional program lines can be inserted into these gaps at a later time without having to renumber the other program lines. Most BASIC programmers use the numbers 10, 20, 30, 40, and so forth in numbering their lines.

Special Note: When working with **array** or **subscripted** variables, (see Chapter 8) **do not** leave a space between the variable name and the opening parentheses in front of the subscript number or letter. Examples:

Always do it this way!	**Never do it this way!**
10 DIM A(10)	**10 DIM A (10)**
20 IF A(X)=5 THEN PRINT Y	**20 IF A (X)=5 THEN PRINT Y**
	These spaces are "no-nos"!

If you ignore or forget this note and leave the "no-no" space, the computer probably will ignore the parenthesis between the subscript variable and the variable used to name the matrix or array. You will get a message from the computer that you may not like. (It will tell you that you have caused an ERROR and you had better mend your "errant" ways!)

"Dummy" Variables: As we saw in Chapter 1, a **function** is a "built-in" computation or expression. Two other items are needed to make a function into a useful statement: First, we need to tell the computer what it is to **do** with the value returned by the function. Normally a BASIC keyword like **PRINT** or **LET** is used for this. Second, the function must have some value on which to operate. This value follows the function call keyword **immediately** and **in parentheses**. For example, the following statement is a typical use of the **RND** (random) function:

```
10 LET X=RND(0)
```

As we shall see when we discuss functions in Chapter 6, the 0 in this case is called a "dummy" variable because any value can be entered without changing the value returned by the random (**RND**) function. The only rule is that **some** number **must** be used following **RND** in parentheses. **RND**(1000) is exactly the same to the computer as **RND**(0).

Important Note: All BASIC statements that will **RUN** in Deferred Mode will also work fine in Direct Mode and vice-versa.

2.3 And now...Let's Continue Our Journey into BASICland!

We'll begin by discussing the commands (and one function), that are used for the creation and execution of BASIC programs on ATARI Personal Computers. These commands, and the function **FRE**, do not alter the program itself, but enable the programmer to use and control the program he has created. We can say that the commands **LIST, RUN, NEW, CONT**, the function **FRE**, and the **REM** keyword are primarily for the benefit of the programmer rather than the computer. They allow **him** to control the computer and the program, and to get information that may be valuable to **him** but not to the computer...

Note that each of the six keywords mentioned above will be discussed using a **format** specificaton, (if you get in trouble, review the Notation Conventions section in Chapter 1), the **abbreviation,** if any, that may be used to save you typing time, one or more **program examples** that show the keywords the way they might appear in a real program, a **description** of what the keyword does and how it might be used, and finally any special rules that must be followed when using the keyword.

Any time you're ready... let's dive right in... it's not really *too* deep!

RUN

Format: **RUN** RETURN (**Note:** no lineno is used—but see also
Program Example: **RUN** RETURN Chapter 5!)

Description: The command **RUN** is used to cause the computer to begin execution of a program. Since it is usually used in the Direct Mode without a line number, the command is executed immediately upon pressing the **RETURN** key. All variables are set to zero, and all files and peripheral devices that were **OPEN** are **CLOSE**d. All arrays and matrices are eliminated (not reset to zero). Unless **TRAP** is used, should any error be discovered while the program is **RUN**ning, an error message will be printed on the screen, and execution will **STOP**. (See Appendix C for a list of the error codes used with your ATARI Personal Computer System). Like the other commands, **RUN** can be used during a program (as part of a line with a number) in Deferred Mode. This would be accomplished in the following manner:

Format: **lineno RUN** RETURN (see Chapter 5 for more info on "chaining" programs)

```
5 PRINT "OVER AND OVER AGAIN" RETURN
10 RUN RETURN
```

NOTE: To begin a **RUN** at *other* than the first program line number, use **GOTO lineno** RETURN.

NEW

Format: **NEW** RETURN (direct mode)
Program Example: **NEW** RETURN (direct mode)

Description: This command erases everything stored in RAM memory! Therefore, be sure you **SAVE** or **CSAVE** (see Chapter 5 on Input/Output operations) any program you may wish to recover and **RUN** later! The **NEW** command is used when you are done with an old program and wish to enter a new one.

Be careful with **NEW** *or you may lose a valuable program!*

CONT

Format: **CONT** RETURN
Program Example: **CONT** RETURN (mode used)

Description: Typing **CONT** and hitting the RETURN key causes execution of a program to resume after the BREAK key has been hit or the program has encountered a **STOP** statement. Execution resumes at the **next sequential line number** following the statement in which the program was stopped. **Note that if the statement interrupted by the BREAK, STOP, or END command is followed by other statements on the same line, the rest of the statements on the line will not be executed when the command CONT is used.** Execution resumes on the **next numbered** line.

If a BREAK is received during execution of the early part of a looping routine (see Chapter 3), and **CONT** is used to continue the program, since the execution is restarted on the **next program line** it is possible that the loop will not be terminated properly and errors may result later in the program. (See **POP**—Chapter 3). For example:

100 INPUT A: (BREAK occurs here.) NEXT I

FRE

Format: **FRE(dummy)** RETURN
Program Example: **PRINT FRE(ϕ)** RETURN

Description: Normally used in Direct Mode, this function is used to get the number of bytes of free (unused) RAM memory. Like all functions, some other command must be used to make use of or to display the value "returned" by the function. In the case of **FRE, PRINT** is most often used. For example:

PRINT FRE(Ø) RETURN

would cause the computer to display the amount of memory available (unused) at the time the RETURN key is depressed.

Rules: FRE, like most functions, may be used in Direct or Deferred Mode; however, it is seldom used within a program.

REM

Format: **lineno REM followed by text of remarks...** RETURN
Program Example: **100 REM ROUTINE TO CALCULATE X** RETURN

Description: REM and everything on the same line following **REM**, is ignored by the computer, (it is a no operation statement). However, when the programmer **LISTs** his program, the **REM** statement is **LISTed** along with all other numbered lines. This gives the programmer a place to make a self-prompt note, or a **REM**ark about the program or part of the program. Obviously, **REM** statements are invisible unless the program is **LISTed**, and the computer doesn't even notice their presence.

Rules: Don't try to put other statements on a line following a **REM** statement! Since the first word on the line is **REM**, the computer simply passes along to the next line number, ignoring everything that follows the word **REM**!

LIST

Format: **List [filespec] [lineno1[,lineno2]]**
Abbreviation: **L.**
Program Examples:

LIST 10,100	RETURN	(prints program list from line 10 to line 100, inclusive)
LIST	RETURN	(prints entire program list)
LIST 10	RETURN	(prints source program line 10 only)
LIST "P:", 10,100	RETURN	(list lines 10 through 100 on printer)

Description: **LIST** displays the source version (as you typed it) of all lines currently in memory, or just those lines you request with the **LIST** statement. For example, **LIST 100** RETURN will cause the computer to display line 100 only. **LIST 100, 150** RETURN will bring you a printout of lines 100 through 150 on the screen. The statement **LIST** used without line numbers following it will cause the entire program to be **LIST**ed.

Rules: **LIST** may be used in Deferred Mode but seldom is. You **might** want to use it in that mode as part of an error trapping routine, though, and you should be aware that it can be so used. (See **TRAP** in Chapter 3.)

The above are commands and functions that will help you steer your way around BASICland, and get your ATARI Personal Computer to give you information you may need in your navigation. You now know how to **RUN,** BREAK just push the BREAK key), **CONT** (continue a stopped program), **LIST** (display what you have entered), check your remaining memory capacity with **FRE**, clear all memory with **NEW**, and make notes in your "trip diary" using the **REM** statement. We've come a long way into deep, dark BASICland!

This would be an excellent time for beginning computerists to review Appendix F, which fully describes the use of the keyboard. See Appendix A, also. Type in a sample program or two to get you started. When you're ready for another BASICland adventure, just turn the page and take a tour of the keywords used for program sequence control and branching!

CHAPTER 3

PROGRAM SEQUENCE CONTROL AND BRANCHING

Ready for more treks through BASICland, huh? Got to admire your perseverance! This chapter will make things easier for you. Matter of fact, each little tour you complete by studying a chapter in this book makes the next excursion less trying and more fun!

This chapter's a little like the steering mechanism on our tour bus. The statements covered will give you control over your program and allow you to tell it where to go next.

3.1 Branching

Remember that the BASIC program follows a sequence of line numbers, in numerical order? That's true, but also remember that the only design function of any user-programmable computer is to do what **you**, the **Master**, tell it to do. So there must be some way to alter the sequence of operation... **and there is!**

In fact, there are two general methods, both of which can be lumped under the generic name "**branching**". **Branching** simply means jumping backwards or ahead any number of lines and transferring control of the program to the line you specify. For example, if we are at line 100, and we want to repeat a computation or other action back at line 50, we can tell the computer to do just that: Go to 50. Likewise, we can tell it to skip certain program lines and **branch** forward (over them). (Actually, **all** statements and keywords in BASIC must be typed into the computer in all caps, and the word is shortened by removing the space between "go" and "to".) The result is the **GOTO** statement which we will discuss in this chapter. But you get the idea of **branching**. It just means going back to an earlier line number of skipping forward over some intervening lines to the one specified.

In general, we can speak of two major types of branching: These are the **unconditional** kind (like the **GOTO** statement we mentioned briefly above), which will happen under any circumstances, and the **conditional** type of branching which occurs only **IF** certain conditions are met.

3.2 Down the Loop Trail...

Before we begin our discussion of the various branching statements themselves, we need to discuss one more concept. This is the idea of **looping**. By use of an **unconditional** branch statement (like **GOTO**), we can cause the program to branch back to an earlier line number, repeat the part of the program between the line number and the **GOTO** statement, branch back again, etc. The reason this is called **looping** becomes rather obvious at this point! Any statements or commands contained in lines between the **GOTO** statement and the "target" line number will be repeated each time the program makes another trip through the loop.

This looping is not only a very powerful programming tool, it enables the computer to do many of the repetitious and menial tasks that a human used to have to do by hand.

NOTE: A loop started and controlled by an unconditional branch statement such as **GOTO**, once **RUN**ning, can be exited only by either powering down the computer (and consequently losing your entire program), or by hitting the BREAK key (gently, please!) The latter simply stops execution, which may be restarted simply by executing either **RUN** or **CONT.**

Here is a Simple Looping Program:

```
10  X=X+1                    RETURN
20  PRINT "X= "; X           RETURN
30  GOTO 10                  RETURN
RUN                          RETURN
```

(When you get tired of seeing it count, press the BREAK key)

3.3 Branching and Program Control

As we discussed, **unconditional branching** is a skipping backward or forward over part of the program to a specific line number (which we will call the "target" line number). Control of the program is transferred directly to the target line number by the branch statement, and execution continues from the target point. This means that the entire line beginning with the target line number will be executed immediately after the branch statement is encountered.

Very Important Note: If multiple statements must be executed on the line that carries the branch statement, the branch statement **must** be placed as the last statement on that line. Likewise, the target line **must not** carry any statement that you don't want executed when the branch occurs!

Since some of the statements discussed in this section cause unconditional branching, a method must be found to "escape" from the resulting program loop, even if it is only by pressing BREAK.

GOTO

Statement:	**lineno GOTO aexp**		(aexp evaluates to the
Abbreviation:	**G.**		"target" line number)
Program Example:	**100 GOTO 50**	RETURN	
	500 GOTO (X+Y)	RETURN	

Description: The **GOTO** statement immediately transfers program control to the target line number. The target may be anywhere in the program, but an embarrassing error will result if the program is told to **GOTO** a line number that doesn't exist! Statements between the **GOTO** statement and the target line number are ignored. Statements on the same line as a **GOTO** but following it will **not** be executed. Any backwards branching **GOTO** statement may cause a loop to be started. If this is unintentional, or if you desire to stop the loop, just hit the BREAK key. Note that a **conditional** branching statement such as the **IF/THEN** statement can be used to break out of a **GOTO** loop. This will be discussed a little later on.

GOSUB and RETURN

Statement:	**lineno GOSUB aexp**	(aexp is the target)
	lineno RETURN	(returns to main program)
Abbreviations:	**GOS.**	
	RET.	
Program Example:	**100 GOSUB 2000**	
	2000 —	subroutine starts on
	2120 RETURN	line 2000

Description: These statements allow the use of **subroutines** in BASIC. A **subroutine** is a program or routine used to compute a certain value or do other work. It is generally used when an operation must be repeated several times using the same or different values. Thus, instead of burning up our own typing time (not to mention the computer's memory space), we can put in a subroutine **once** and then "call" it repeatedly using **GOSUB** (short for go to **subroutine**). The **first line** of the subroutine is the target line specified with the **GOSUB** statement. The **last line** of the subroutine **ends** with the **RETURN** statement, which sends the program back to the next line number **following** the line on which the **GOSUB** statement occured. For this reason, multiple statements that follow **GOSUB** on the same line would **not** be executed.

Programmers should note that when using subroutines, care must be taken to avoid having the main program reach its last statement and then continue executing a subroutine because that subroutine's numbers are sequentially "next" in the program. The easiest way to prevent this is to get in the habit of *always* placing an **END** statement immediately before your subroutine's first line. For example, if the subroutine starts on line 2000, line 1999 should look like this: **1999 END RETURN**

Here's a short program that will illustrate the use of subroutines:

```
5 REM EXAMPLE USE OF GOSUB/RETURN
10 X=100                                    RETURN
20 GOSUB 1000                               RETURN
30 X=120                                    RETURN
40 GOSUB 1000                               RETURN      Main Program
50 X=50                                     RETURN
60 GOSUB 1000                               RETURN
999 END                                     RETURN
1000 Y=3*X                                  RETURN
1010 X=X+Y                                  RETURN
1020 PRINT X,Y                              RETURN      Subroutines
1030 RETURN
RUN                                         RETURN
 400          300
 480          360
 200          150
READY
```

Note: See Section 3.4 for important information on the **GOSUB/FOR** stack and the **POP** statement!

In this example, the subroutine is called three times to compute and print out different values of X and Y. Generally, a subroutine can do anything that can be done in a program. It is used to save memory and program-entering time and to make our programs easier to read and "debug".

FOR, NEXT, and STEP

Format: **lineno FOR avar = aexp1 to aexp2 [STEP aexp 3]**
lineno NEXT avar (avar same as avar above)

Abbreviations: **FOR: F.**
NEXT: N.
STEP: none

Program Examples:

```
100 FOR X=1 TO 10 STEP 2          RETURN
110 NEXT X                        RETURN
100 FOR INDEX = Y TO 100*Y STEP Z RETURN
110 NEXT INDEX                    RETURN
```

Note: STEP aexp3 is optional and defaults to 1 if not used.

Description: The **FOR/NEXT/STEP** statement is used for one purpose only: It sets up a loop and determines how many times that loop will be executed. The loop variable (avar) is initialized to the value of aexp1. Each time a pass through the loop is completed, the **NEXT** statement is encountered, the loop variable is incremented, and if there are more passes to be made, the program control is transferred back to the start of the loop. The value of the loop variable (avar) is increased by the value of the **STEP** (aexp3). The loop then runs again, encounters the **NEXT** statement, etc. Eventually, the value of the loop variable will exceed the value of aexp2. When this happens, the looping stops. The program then executes the statement immediately following the **NEXT** statement; it may be on the same line or the next sequential line.

Other Rules: The **STEP** statement is optional, and should be used only if it suits your particular programming needs. If **STEP** is not used the computer will automatically assign a value of 1. Thus it will count sequentially if you don't tell it otherwise. (See examples below). Something called a "**GOSUB/FOR** stack" is used to keep track of the looping operation. This "stack" is really a group of memory locations. Information can be placed on top of the stack, (pushing) and pulled off again (poping). When a **FOR/NEXT** loop is terminated the loop information is popped, thus clearing the stack for use in another loop later in the program. It is not essential that the beginning "tourist" understands this bit of BASICland flora/fauna. But it isessential to know that if an "unnatural" departure from

the loop is made (by a branch statement) the loop information **remains** on the stack, and future errors may result! See the description of the **POP** statement in Section 3.4. It will tell you how to **POP** your stack so the computer won't "blow it" later!

Here are some examples of the **FOR/NEXT/STEP** statements at work:

```
READY
10 FOR A=1 TO 100      RETURN    (Pass through loop 100 times)
20 Z=Z+A               RETURN    (Add loop number to previous total)
30 NEXT A              RETURN    (See if any more loops to do: If so, do
40 PRINT Z             RETURN    one: If not, proceed to next line)
RUN                    RETURN    (Print out total of all numbers between
                                 0 and 100 inclusive)
5050                             (Answer is 5050: 1+2+3...100)
READY                            Comment: Try that on your $9.95 hand-
                                 held calculator, Chum!
```

Now change line 10 only by adding the words "STEP 2":

```
10 FOR A=1 TO 100 STEP 2    RETURN
And RUN it again!
READY
RUN                         RETURN
2500
READY
```

What has it done? Well, since it started with 1, then added 2 to each pass through the loop, the second pass must have been with A = 3, the third pass with A = 5, etc. The last pass had A equal to 99. So it added up all the odd numbers between 1 and 100 (inclusive)! If you don't believe this, add the statement **PRINT A; " "** (note semicolon is important here) at line 25. Now it will print out all the values of A as it encounters them! And they were all the odd numbers between 1 and 100, right? Here's what the program change was and a typical RUN:

```
Add to Program...
25 PRINT A; "  ";    RETURN
```

Note: The trailing semicolon causes results to be printed in lines instead of one value on each line. The quotation marks enclose one blank. This provides one space between the numbers.

```
RUN                                                    RETURN
1  3  5  7  9  11  13  15  17  19  21  23
25  27  29  31  33  35  37  39  41  43
45  47  49  51  53  55  57  59  61  63
65  67  69  71  73  75  77  79  81  83
85  87  89  91  93  95  97  99  2500
READY
```

(See, it did add up only the odd numbers! And it stopped at 99 because adding 2 as called for by the **STEP 2** statement would have put it above the 100 called for in the **FOR** statement. Pretty smart of it to resolve the "conflict" all by itself!)

Here's a handy tip: Occasionally, you may want to run a **FOR/NEXT** loop backwards (say, from 50 to 20). The statement for this would be

```
100 FOR X=50 TO 20 STEP -1          (note minus sign)
```

And another: STEPs can be decimals or fractional numbers. Like this:

```
100 FOR Z=1 TO 100 STEP .789
```

Nesting Loops: We'll conclude this little tour of the unconditional branch statements and the resulting loops by describing how one loop can be "nested" inside another for some REAL computing power!

Would you believe we can print out every possible multiplication problem that will result in an integer (whole number... no decimals) product between 1 and 100, with the answers? And do it with a program that has only 5 lines and *no* multiple statements on any line? We can with nesting! Here it is:

```
READY
10 FOR A=1 TO 10                    RETURN
20 FOR B=1 TO 10                    RETURN
30 PRINT A;"*";B;"=";A*B;" ";       RETURN
40 NEXT B                           RETURN
50 NEXT A                           RETURN
READY
```

Now **RUN** it and gaze in amazement as the computer does what you told it to do! First it says to itself "A=1", passes to line 20, sets B equal to 1 also, goes on to line 30, does the multiplication and printout, and proceeds to line 40 which tells it to take the next B. So it branches back to line 20, increments B to 2, goes to line 30, does the multiplication and printout... etc., etc.... **until it finally gets to the last B value (10).** At that point, instead of branching from line 40 back to line 20, it proceeds to 50, which tells it to take the **NEXT A**(A now equals 2). Now the B loop starts again, from 1 to 10. Run this one a few times, examine the screen carefully, and note how this nesting procedure works.

You can "nest" any number of loops, one inside the other, subject only to memory limitations and execution time requirements. The only hard and fast rule is that you must make sure the nesting is complete... that is, each loop must be completely and totally inside the next one. This is done by making sure that the **NEXT** statements are proper: Each loop is closed before the one outside of it is. The little drawing below shows a better "picture" of the the nests and how they fit together.

NESTING OF FOR/NEXT LOOPS

```
10 FOR A=1 TO 10
20 FOR B=1 TO 10
30 PRINT A;"*";B"=";A*B;"   ";
40 NEXT B
50 NEXT A
```

OUTER NEST "A"
(Opened first, closed 2nd)

INNER NEST "B"
(Opened second, closed 1st)

SOME MISCELLANEOUS COMMENTS ON FOR/NEXT LOOPS

- Every **FOR/NEXT** loop is executed **at least** once, even if aexp1 is greater than aexp2!

- When the **STEP** size (aexp3) is less than 0 (i.e. is negative), then the loop terminates when aexp1 is **less** than aexp2.

- The value of aexp2 is stored on the **GOSUB/FOR** stack, and is evaluated once (when the loop is entered).

- The loop variable (avar) may be changed during the loop. To terminate a loop prematurely, for example, you can use an **IF/THEN** or other conditional statement to reset avar so that it is greater than aexp2 if the condition is met. The loop will then end and the program will continue with the line following the one on which **NEXT** appears.

- Any time the loop variable becomes larger than aexp2, the loop will stop, and control passes to the line following the one on which **NEXT** appears.

That concludes our "circular tour" of **FOR/NEXT** loops and nesting. Let's take a break now, and after our (optional) lunch, we'll take a tour of the **conditional** branch statements!

3.4 Conditional Branch Statements: General

A **conditional** branch statement is much like the unconditional kind we discussed earlier but can be **much** more powerful. A conditional branch occurs if (and only if) certain conditions are met. These conditions may be either logical or arithmetical—The computer doesn't care which, for it evaluates everything arithmetically anyhow! If the arithmetic expression (aexp) that follows the branch statement keyword evaluates to a logic zero (false), control passes to the next line number in sequence, and **nothing** on the conditional branch line is executed. If the expression evaluates to anything other than zero (true), then the statements on the rest of the line **are** executed. These statements may be used to send the program off to another line (branching), or they may be executed as multiple statements on the same line as the conditional statement keyword and expression. This will become clearer with a little thought and a few examples. First, we'll examine the **IF/THEN** statement.

IF/THEN

Statement: lineno IF exp THEN statement2 [: statement3]...
Abbreviations: None
Program Examples:

```
100 IF X=100 THEN 150                                  RETURN
100 IF A$="KATY BAR THE DOOR!" THEN 190                RETURN
100 IF AA=145 AND BB=1 THEN PRINT AA, AB               RETURN
100 IF X=100 THEN X=0                                  RETURN
```

Description: IF/THEN works exactly as we described in our discussion of conditional branching in general. The computer evaluates the branch condition expression (called "exp" in our statement format) to see whether it is equal to zero (false) or to any other number (true). If you are a beginner, don't worry at this point how it does this... just accept it for what it is: A miracle! If the condition expression is met (true, or "non-zero") the program carries out the branch or other statement following **THEN**. If the condition expression is false ("0"), control simply passes to the next line number in numerical sequence as usual.

OK, we now know all about **IF/THEN**, realize it is a powerful tool, and that it will be something we can surely use someday, right? Let's convert that feeling **right now** into one that says "Hey, that's fabulous! I'll use that in every program I ever write!" Maybe an example program or two will do the job...

Prime Number Generator Program

We've finally gotten to the point where we can start having a little more fun with our program examples! This one not only shows the IF/THEN statement at work, it also demonstrates the "computer approach" to problem solving, and the incredible "thinking power" of the computer that comes with the ability to loop back to an earlier part of the program and repeat calculations. This gives the machine the capability to run trial and error solutions to problems: Try a calculation, test it for acceptability, and if not within the programmed parameters, reject it and try another calculation! Now keep doing it until all the correct solutions are found and printed out.

Prime Number Generator

Note: A prime number is any number greater than one that has no factors except itself and one. Obviously, two is the lowest prime number (factors are itself and 1). 4 is the lowest positive number that is **not** prime (factors are 2 and 2). 5 is prime, 6 is not (factors 2 and 3) and so forth. But let's go get our computer to do the work...

```
10 GR. 0                                          RETURN
20 PRINT "  PRIME NUMBER GENERATOR"                (6 spaces after first ")
30 PRINT:PRINT "START GENERATING AT W
HAT NUMBER?"
40 INPUT X
50 IF X < 2 OR NOT X = INT(X) THEN GOT
O 10
60 IF X=2 THEN PRINT "2"
70 IF X/2 = INT(X/2) THEN X=X+1
80 FOR Y=2 TO SQR(X)
90 IF X/Y = INT (X/Y) THEN GOTO 120
100 NEXT Y
110 PRINT X
120 REM SKIP EVEN NUMBERS
130 X=X+2                                         RETURN
140 GOTO 80
```

Here's how the Prime Number program works: Line 10 places the computer in Graphics Mode 0. When you look at Chapter 9 you'll see that this is the normal text mode and that without another **GRAPHICS** command, the system "defaults" to Mode 0 anyhow. So why tell it to go into Mode 0? Simple! Every time we tell it to change modes, it clears the screen! So this is just a super-slick way of clearing the screen at the start of our program. Lines 20-40 title the program on the screen and ask you to input the number where you want the program to start looking for primes. Note the 6 spaces following the first " in line 20. This is to center the title. Note that the variable X is used to store the number you input as the starting point. Line 50 simply says that if the starting number is less than 2, start the whole program over! That's to catch faulty input. If "human error" is present (the starting point is less than 2), the computer now craftily starts over and asks you to enter the data again...but without the usual embarrassing **error** message! Line 60 just tells the computer to print out the known fact that the number 2 is prime... but only if your starting point is 2.

We know that an even number cannot be prime, for it is always evenly divisible by 2, and 2 is thus a factor. Therefore, line 70 says that the computer is to increment (add one) to the first number being tested if it is **even**. Note that if a number divided by two is the same as the INT function of that number divided by two, the original number MUST be

even! (See Chapter 6 for a description of the **INT** function, which simply rounds any number with a decimal fraction **down** to the next lower whole number or **integer**.

Line 80 starts a **FOR/NEXT** loop that will run from 2 to the square root of the number you told it to start with in line 40. Line 90, which is in the loop, tells the program to check X/Y and INT(X/Y) to see if this division "comes out even". If so, then the program jumps to line 120. If the division does not "come out even", then Y is not a factor, and the loop is run again with a different value of Y.

Once all the possible values of Y (from 2 to the square root of X) have been checked, the program arrives at line 110, having determined that there are no factors of the number stored in X. This means that the number must be prime, so line 110 tells the computer to print out the number. Line 120 is a remarks (**REM**) line that tells the programmer that this is the part of the program that weeds out all even numbers. Since the number printed at line 110 is prime, it must also be odd! So incrementing X by 2 means that the value of the number stored in X is now the next consecutive odd number higher than the last prime discovered. The program is now directed by line 140 back to line 80, the new value of X (which we already know is an odd number) is checked for prime, and so forth...

The program will just keep on grinding out prime numbers "until the cows come home", the power fails, or (most likely) you get tired of watching it.

Ah, the awesome power of Looping! (Didn't he used to play for the Dallas Cowboys?)

ON/GOTO
ON/GOSUB

Statement: **lineno ON iexp GOTO/GOSUB lineno [, lineno...]**
Abbreviations: **None**
Program Examples:
```
100 ON X GOTO 200, 300, 400, 500
100 ON A GOSUB 1000, 2000
100 ON SQR(X) GOTO 30, 10, 100
```

Description: The **ON/GOTO** statement is much like **IF/THEN** but with more power added. As you can see, there is a list of line numbers. There is also an expression called "iexp" which means an arithmetic expression that evaluates to a number which is then rounded

to the nearest positive integer (whole number) value up to 255. If the resulting number is 1, the program control is transferred to the first target line number in the list. If it is 2, control goes to the second number, and so forth. Very powerful stuff indeed! If the number evaluates to zero or is greater than the number of line numbers in the target list, then the test fails and control passes to the next sequential line number. As with other **GOTO** or **GOSUB** statements, multiple statements following **ON/GOTO** or **ON/GOSUB**on the same will *not* be executed.

Here is a little routine that will demonstrate **ON/GOTO:**

```
10 X=X+1                                                    RETURN
20 ON X GOTO 100, 200, 300, 400, 500
30 IF X=5 THEN PRINT "THAT'S ALL FOLKS! MY JOB IS DONE
!"                                                          RETURN
40 GOTO 10                                                  RETURN
100 PRINT "NOW WORKING AT LINE 100": GOTO 10                RETURN
200 PRINT "NOW WORKING AT LINE 200": GOTO 10                RETURN
300 PRINT "NOW WORKING AT LINE 300": GOTO 10                RETURN
400 PRINT "NOW WORKING AT LINE 400": GOTO 10                RETURN
```

Now "RUN" it and see what happens. Sure works fast, doesn't it!

TRAP

Format:	**lineno TRAP aexp**	(aexp is target line number)
Abbreviation:	**T.**	
Program Example:	**100 TRAP aexp**	

Description: The **TRAP** statement is used to direct the program to a specified line number if an error is detected. Without the **TRAP** statement, when an error is discovered execution stops and an error message is printed. This can be annoying, to say the least, when you are inputting data into your computer, make a typo, hit RETURN and the thing comes up with a rude message. The worst part, though, is that you would then have to **RUN** the whole program again, and make all those typing inputs "just one more time!" **TRAP** can prevent all that bother...

By inserting a **TRAP** statement before each **INPUT** statement, and telling the machine that if an error is made it should go to a certain line and ask for the input to be repeated, it will simply ask you for a repeat of what you "blew" the first time! Here's an example routine:

```
10 TRAP 20          (sends program back to 20 in case of error)
20 INPUT X          (See Ch. 5) (prompts you to enter a number)
30 PRINT            "THE VALUE OF X HAS BEEN SET AT ";X
```

NOTE: Since you're getting to be quite a programmer now, we will stop "prompting" you to hit the RETURN key after each line has been typed in. You will forget a few times, but hopefully you are in the habit of doing it by now...

First **RUN** this little program, entering the number 2 (don't forget RETURN). If all goes well, try entering a letter rather than a number (this would result in an error, as the variable X is a numeric variable, *not* a string variable.) Gave you the old "?", didn't it? Well, it's better than an error message any time!

Rules: The **TRAP** statement works on any error that may occur *after* the statement. But once an error has been detected and **TRAP**ped, we must "reset" the trap with another **TRAP** statement. For this reason, if you want to **TRAP** errors every time, it is sensible (memory capability permitting) to **TRAP** each statement that exposes your computer to "human error" possibilities. The statement you **TRAP** to could be used (in part) to reset the **TRAP**!

Note, too, that it is NOT necessary to "TRAP" directly back to the input line. For example, you could send the program off to a little routine that prints a humorous message like "Whatsa wrong wit' you, you just nacherly dumb?" or some such *and then* (with the **GOTO** statement) returns the program to the input line. You could also **TRAP** back to the **TRAP** statement, thus resetting the **TRAP**! Lots of possibilities... use your imagination! After all, that's all you have between those ears that your computer doesn't have better and faster in its electronic innards!

That concludes our little tour of the program sequence controls available to you in ATARI BASIC. With the six statements we have discussed you can do anything you'd like in the way of directing your program. We'll pick up three more program control statements next, as these help us allocate memory and organize our programs for best execution at "RUN time".

POP

Format:	**lineno POP**	(this is a real toughie!)
Abbreviation:	**None**	
Program Example:	**1000 POP**	(see?)

Description: We discussed this briefly before, and the mechanical/electronic workings of the computer are beyond the scope of a software book anyway. But remember that "stack" of memory locations we mentioned? All you really need to know is that the number of loops to be executed, and the **RETURN** target line in a **GOSUB**, are controlled by the top entry in the stack. Thus, if a **GOSUB** is *not* terminated by the execution of **RETURN**, or a **FOR** statement is not followed by an executed **NEXT** statement, the top memory location on the stack is still "loaded" with some number. In case another loop or **GOSUB** is executed, that top location needs to be cleared: The data byte there needs to be wiped out. We call this "**POP**ping" it off the stack. (The opposite, also mentioned earlier, is called "pushing" data onto the stack.) Naturally enough, the statement **POP** is used to clean up our stack and prepare it for the next loop or **GOSUB**.

Rules: **POP** must be used according to the following rules:

1. It must be in the execution path of the program.
2. It must follow the termination of any **FOR/NEXT** loop if the program exits the loop without completing the number of loops called for by the **FOR** statement; in other words, the exit is made by a separate branch statement. (See example program below.)
3. It must follow the execution of any **GOSUB** statement that is not brought back to the main program by a **RETURN** statement.

Perhaps a couple of examples will help clear up the confusion for you:

```
10 FOR INDEX= 1 TO 100
20 PRINT "THIS IS A FOR/NEXT LOOP WIT
H INDEX = "; INDEX
30 IF INDEX = 10 THEN GOTO 50
40 NEXT INDEX: END
50 PRINT "THIS LOOP WAS PREMATURELY T
ERMINATED & NEEDS 'POP'"
60 POP
```

As you can see, the program goes through the loop 10 times and then is forced to exit by the **IF/THEN** statement on line 30. The **FOR** statement called for 100 passes through the loop, and this unnatural exit calls for use of **POP**.

Here's an example of **POP** used with **GOSUB** (without the **RETURN** executed):

```
10 GOSUB 1000
15 REM LINE 20 WILL NOT BE EXECUTED
20 PRINT "NORMAL RETURN PRINTS THIS ME
SSAGE":END.
30 PRINT "ABNORMAL EXIT FROM SUBROUTIN
E PRINTS THIS"
40 POP
999 END          (prevents program from "crashing" into subroutine)

1000 PRINT "NOW EXECUTING SUBROUTINE"
1010 GOTO 30
1020 RETURN
```

In this example, line 10 sends the program to the subroutine at 1000. At that point, the computer prints the subroutine message, passes on to 1010 which "abnormally" exits from the subroutine, going back to the main program. **Note that the RETURN statement on line 1020 is never executed,** and this means that we need the **POP** statement to "clean up" stack operations in case future loops follow. The main program is re-entered at line 30, which prints the "ABNORMAL EXIT" message, passes on to the **POP** statement on line 40, and finally **END**s on line 999 (thus preventing re-execution of the subroutine).

Looking back on our little trip, this chapter has brought us a long way. Your tour is not yet complete, though, for there is much more to know and see. Hope you've enjoyed yourself so far!

DANGER! MATHEMATICS AHEAD.

CHAPTER 4

MORE ON VARIABLES AND MATHEMATICAL MONKEY MOTION

Welcome back to your guided tour of BASICland! Today we'll be in jungle country. We'll shoot the rapids on our Floating Point, have another look at variables, and pick up a few glimpses of computerized mathematical monkey motion. We'll see the order that the computer uses to conduct evaluations of expressions (depending on the **operator** or sign used). By the time our little side tour into the Numerical Wilderness is complete, we'll be ready for tomorrow's jaunt into the Land of Inputs! Got your cameras and notebooks ready? Let's begin.

4.1 Number Crunching: How Floating Point Numbers are Manipulated

Floating Point is really a short way of saying floating **decimal** point. That means that ATARI Personal Computers can take care of decimal points without you having to worry about them. BASICally, you can enter numbers into the computer by simply putting them in the keyboard in a normal way (meaning a **human** way). Simply type 3.4567, and the computer will track the decimal point for you. You can tell it to add, subtract, multiply, and divide, and it will locate and handle the decimal point.

The floating decimal point arithmetic can handle numbers between ±9.99999999E+97 and ±1.0E−98, including of course, zero. Now all we have to do is learn how to transfer our own thought process into the kind of scientific notation the computer uses. For those of you who already know scientific notation, 1.456E+3 is exactly the same as 1.456×10^3. So much for you smart guys! For the rest of us, let's take a more down to earth approach. The actual digits or figures we need worry about are the ones before the E. In the above example, these are 1456. But where does the decimal point go? In scientific (or computer) notation, it always goes after the first digit of the figure we are dealing with. In this case, our number is 1.456. The E stands for "exponent" which just means that power of ten that we must multiply the basic number by to get the decimal point in the right place. Again, let's stress that the computer will do this for us, and all this is by way of explanation of how it works. Saying E+03 is exactly the same as saying "*10^y" or "times ten to the third power". Let's try it.

We know that our actual figure is 1.456, and that the exponent (E) is +03. Since ten raised to the third power is a thousand (10*10*10), we must move the decimal in our number just as if we had multiplied it by 1000. This means move it to the right three places. We get 1456. Now let's tell the computer (in direct mode) to **PRINT 1.456E+Ø3**. Betcha it printed 1456! It did if you remembered to hit RETURN.

The computer can take an entry in computer notation and translate it into our system so we can understand it easily. But there are limits to everything! If you ask it for more than 9 significant figures it will "lapse" into scientific notation. For example, if you tell it to **PRINT 15000000000**, it will give you back 1.5E+10. This means you really ought to know what those numbers mean. There's a simple rule, and here it is:

To convert scientific notation to conventional notation, move the decimal to the right, if the exponent is positive, the number of places shown by the exponent. Move it to the left if the exponent is negative.

Examples:

1.34E+12 is the same as 1340000000000.0
2.4359E−10 is the same as 0.00000000024359

More experienced readers will be interested to learn that a special kind of binary coded decimal storage is used by ATARI Personal Computers. The exponent is stored separately from the digits in the actual number, which are stored in hexadecimal notation (but the hex numbers **represent** decimal numbers, so we can **PEEK** into the location a number is stored in, and "read" the number in decimal form.) This can be a powerful debugging tool. (See Chapter 6 for description of the **PEEK** statement.)

4.2 On the Nature and Naming of Variables

Way back in Chapter 1, we discussed **variables**. A few more words on the subject are in order at this point. Think of a variable as a **place to put data**. Each time we equate (**LET** it equal) a variable and a number, string, or whatever, we are really storing that string or number in a location represented by the variable's name. As you can see, this must only be done once, (unless the value of the variable is to be changed) and from that point on, referring to the variable by name is the same to the computer as referring to the data stored in that variable. This may seem unimportant, but it is not. We should develop an automatic thought process in our own minds. We do **not** set the variable A equal to a number! We **store** the number in the location represented by variable A!

Naming variables can raise all sorts of good **and** bad side-effects. It's always nice that we have the **capability** to name a variable with a jawbreaker that's up to 120 characters long, and can mix numbers and alphabetical (upper case) characters, but the first character

must be a letter. The first thing that usually occurs to a beginner is to use memory joggers as variable names. For example, in a checkbook balancing program, we could use CHECK or CHECKNO or OLDBAL. This is a fine idea, as it makes our programs more "human" and easier to **RUN**, debug, modify, and read. Remember, though, that each character used costs us one byte of memory, and that spaces and characters other than upper case (capital) letters and numbers are no-nos (so are blank spaces!). For example CHECK BAL is illegal. (The space will give your computer the shivering fits.)

The second thing that happens is that we start using variables like **GOTO** or **GOSUB** or **FOR** (because they are used on lines that use these BASIC keywords.) This is a sure-fire method of hopelessly confusing your poor computer. **Don't ever use any "reserved" words as variables.** It won't work, so don't even bother to try it. (Reserved words may, however, be used within quotation marks, in **PRINT** statements, and may be used anywhere in a line beginning with **REM**).

NOTE: Appendix B is a list of all Reserved BASIC words, and should be consulted if you think you may be using a Reserved word as a variable. One last word to the wise: **don't!**

Here's an example showing why we don't ever use reserved BASIC words as variable names!

Run this on your own computer. You'll note that the poor thing gets hopelessly confused and can actually be "conned" into giving out a wrong answer: Great shame for a computer!

```
10 B = 0
20 NOTB = 2
30 PRINT NOTB
RUN
1
```

Here's what happened: In line 10, the value 0 is assigned to the variable B. In line 30, the value 2 is stored in the variable **NOTB**. In the **PRINT** statement in line 20, the variable **NOTB** is picked up by the computer as a logical **NOT**. Since B is 0, by computer logic, **NOT B must** be **1**! We have the poor baby mixed up between variable names and logical operators! The computer prints the "1" faithfully.

Lesson To Be Learned: Don't **ever** use variable names that have reserved BASIC words as a part of them, and most especially not if the variable name **starts** with a reserved BASIC word (such as **NOT**).

4.3 Now for the Arithmetic Operators

As we have seen, an **expression** conducts mathematical evaluations which result in numerical answers (yes, this is true of string evaluations, too... more on this in Chapters 6 and 7). A **statement** may also consist of mathematical operators (you are used to seeing them called "signs", but "operators" is a much better word as the computer uses them to see what **operation** is to be conducted). Logic operators also perform mathematical evaluations, but in a somewhat different way. If we state that the sky is blue, and tell the computer that the variable name SKY is equal to string "BLUE", we would have to do it like this. **100 SKY$ = "BLUE"** (note **$** indicates string variable) Now if we came back later and asked it whether the sky is blue, it would evaluate our question, see that the answer is yes, and print whatever we told it to. Try this program:

```
100 SKY$ = "BLUE"
110 IF SKY$ = "BLUE" THEN PRINT "YES, THE SKY IS BLUE"
```

Here's how it does this crafty little trick of logic: Since the variable SKY$ is equal to the string "BLUE", the **IF/THEN** statement in line 110 is evaluated to a 1 (true), and the message is **PRINT**ed. If we had said in line **110 IF SKY$ = "RED" THEN PRINT "YES, THE SKY IS BLUE"** the evaluation would have come up with a zero (false), and the program would have tried to go on to the next line, found none, and quit.

As you can see, these "logic" operations are handled somewhat differently (but not really very much) from pure arithmetic ones. It is worth bearing in mind that computers use numbers (1's and 0's) the way we use letters and words. When we "feed" it words, the computer immediately translates or "interprets" them as numbers, performs the calculations necessary, and then interprets the result back to us as either words or numbers of a type we can use. Remember that in the language of our ATARI Personal Computer, 1 means true, while 0 means false, (at least in most cases).

Addition: The conventional addition sign (+) is used as the operator here. For example:

```
100 A=1+3+6+8          (Add 1,3,6 and 8 together. Store the
                        result in the variable named A)
100 PRINT 2+2          (Add 2 and 2, get 4, and PRINT it)
```

Subtraction: Again, the normal (−) sign is the operator.

Multiplication: The asterisk (*) is used instead of the normal "x" which might be confused by the computer or the programmer with the letter X. Thus, to multiply 2 time 3, we write:

```
100 X=2*3
or
100 PRINT 2*3
```

Division: The slash-mark (/) is used in the "normal" way to mean the division operation. Thus:

```
100 Y = 2/3
or
100 PRINT 2/3
```

The Equal Operator and the LET Statement: Our old friend the equal sign (=) is a bit tricky, as it has two totally separate meanings in BASIC. It may be used with the (optional) **LET** statement to set a variable name equal to a value. This looks like the following:

```
100 LET X = 3.142*16        (Note LET is optional)
```

You should see here that we are not telling the computer that something is really true or equal, but giving it a command to **LET** it be so. That's how we can say apparently illogical things like:

```
100 X=X+1
```

and get away with it. Remember we are **not** discussing a logical operator here; we are storing a piece of information in a location designated for computer reference purposes with a variable name (X in this case).

The **other** use of the equal operator is to perform **logical** or **arithmetic** operations. The **IF/THEN** statement is often used for this purpose:

```
100 IF A=100 THEN PRINT A
100 IF A-B = C-D THEN PRINT "Your Deal"
```

Note that in the latter examples we are **performing** logic **operations**, and **not storing data!**

Exponentiation: The up arrow sign (∧) is used to denote raising a number to a **power** (squaring, cubing, etc.). This is usually done as an expression or as part of one. The number immediately before the arrow (this may be a variable where a number is stored) designates the number to be manipulated, while the number or variable following the up arrow indicates the power to which the first number is to be raised. Thus:

```
100 X= 3  4
100 PRINT 10∧3
```

Just in case you didn't take higher (?) math in high school (with things like exponents and powers): An exponent indicates how many times the number must be multiplied times itself to get the value of the expression. For example:

10∧3 is exactly the same as saying 10 * 10 * 10
(and so is 1000, which is the "real" value).

More Notes on Raising a Number to a Power Using ∧ : The computer's Floating Point arithmetic and the internal methods used to handle it can cause some interesting side effects: Try this:

```
PRINT 2∧3                    (Direct Mode... The answer should be 2³
                              or 8)
7.99999991                   (Computer Prints)
READY                        (this at least is true!)
```

Well, folks, it's not really wrong. And this is probably close enough for government work. And we do have two ways to "beat the house". First, we can tell it to take 2*2*2 and it will be happy to print out "8" for you! An even slicker way is with a bit of **BASIC** programming. Like this:

```
10 PRINT INT((2∧3)+.5)       (Deferred Mode... you can do it in
                              direct, just as well. Try it!)
```

So if it bothers you that there is apparent inaccuracy in the 8th or 9th decimal place, use one of these simple fixes. The world is full of compromises, and even computers aren't perfect!

Comparison Operators: Numbers or strings, and the variables where they are stored, may be compared to see if they are equal, not equal, greater than, less than, greater-than-or-equal-to, or less-than-or-equal-to each other. The operators for these are shown here:

Operator	Operator Description
<	Less than
<=	Less than or equal to
=	Equal to
>=	Greater than or equal to
>	Greater than
<>	Not equal to

Except for =, which we have already discussed, and its two meanings in BASIC, the above comparison operators are very simple and straightforward. They do just what they should do with no tricks as long as we remember that they are **logical operators** and cannot be used to store data in a particular place that is to be identified by a variable name. For example, we cannot state to the computer: 100 A < > 3

What we can and **will** do is use statements or expressions that associate a number with a variable, and then use the comparison operators to compare that variable with others. For example:

```
10 LET A=3                          (store 3 in the location designated by
                                    the variable name A)
20 If A>5 THEN PRINT "A is greater than 5"
30 IF A<5 THEN PRINT "5 is greater than A"
```

Finally, we have the logical operators **AND, OR,** and **NOT.** Again, these are straightforward and simple. They, (except for **NOT**). are used to **compare** two expressions. Here is an example of the **AND** operator at work:

```
100 A=1: B=2
110 IF (A<B) AND (B=A+1) THEN PRINT "THE AND STATEMENT IS TRUE"
```

AND requires that *BOTH* of two expressions be true, in order to evaluate to a "1" (true).

OR is even easier. If either of two expressions **OR** both are true, the **OR** comparison yields a "1" (true). A "0" (false) is the result only if both statements are false "(0)". For example:

```
100 A=1: B=2
110 IF A=1 OR B=1 THEN PRINT "THIS OR STATEMENT IS TRUE"
```

NOT can be a little trickier (but not much!) If **NOT** is applied to an expression which is true, a zero will be the result of the evaluation. If the expression is false, the result will be a "1". Thus:

```
100 A=1
110 PRINT "NOT A="; NOT A
RUN
NOT A=0
```

4.4 Order of Execution of Arithmetic Operations

ATARI Personal Computers do their arithmetic in the following numerical order of priority. Operators on the same line in the Table have equal priorities, and in the case of two or more signs of equal priority, the one on the left (closest to the BASIC line number) takes precedence.

Precedence	Operators	English "Translations"
1	<,<=,=,>=,>,<>	Less than, less than or equal to, equal to, greater than or equal to, greater than, not equal to
2	-	Minus sign: Unary minus
3	^	Exponentiation - Raise a number to the specified power.
4	*, /	Multiplication, division
5	+ -	Plus, minus (used as binary operators)
6	<,<=,=,>=,>,<>	Less than, less than or equal to, equal to, greater than or equal to, greater than, not equal to
7	NOT	Logical NOT
8	AND	Logical AND
9	OR	Logical OR

Note: It is important to understand the above priorities (called precedence) in computer work. Here's a good example of why this is true.

What Will it Print?

```
10 A = 5 + 6*3
20 PRINT A
30 END
```

It will print 23

What Will it Print This Way?

```
10 A = 3*5 + 6
20 PRINT A
30 END
```

This way it will print 21

(The correct answer depends on the programmer!)

However, if the programmer should forget the order of precedence (* and / take priority over + and −), parentheses could be used as follows:

```
W = (X/A) + W*598
```

The next expression, however, will not give us the answer we want. Rather, it will take the *sum* of A and W and *divide* the result into X. That result will then be multiplied by 598.

```
YY = X/(A + W)*598
```

Try evaluating the two above expressions by using the following simple routine on your Atari Personal Computer:

```
10  X=1000
20  A=2
30  W=8
40  YY= (insert the right side of the expression here)
50  PRINT "I WILL ASSIGN THE VALUE ";
YY; "TO THE VARIABLE YY."
```

Change line 40 to experiment with the use of parentheses. Remember that your computer is not solving an equation. It is assigning the value of the expression on the right side of the equal sign to the variable name on the left side. It does this by performing arithmetical operations according to the signs you give it in the expression, in the order that ATARI has told it to use. You can vary that order with parentheses.

As much as we love it, and as wonderful as it is, **you**, the programmer, are the master of your computer, and it will do **only** what you tell it to.

4.5 Precedence Control by Use of Parentheses

The normal sequence of arithmetic operations described above can be altered by use of parentheses ( ). Placing an expression or a piece of an expression inside parentheses causes that expression (or piece) to be evaluated at the top of the priority list for that logical line. If several parentheses are "nested" inside one another, the innermost pair contains the expression that will be evaluated first, and so forth. Here are a couple of examples of use of parentheses:

Example #1: We want to set X equal to the **sum** of 2 and 14 divided by 4. We know that expressions in parentheses are evaluated first, and that division will be carried out ahead of addition. Thus, if we wrote **"X = 2 + 14 / 4"** the computer would first divide 4 into 14 (and get 3.5) and then add 2, getting a value of 5.5 which would be set equal to X. Therefore X would = 5.5. We want it to do the **addition first**, and **then** the division, so we place the addition part of the expression in parentheses like this:

```
X = (2 + 14) / 4
```

Now the computer sees it like this: "First add 2 and 14 and get 16. Take the 16 and divide it by 4, coming up with 4 as the answer."

The beginning computerist should bear in mind that precedence problems most often account for strange results or wrong answers when the program otherwise appears to be running correctly. Learn and use parentheses!

Example #2: Write an expression that will set the variable YY equal to the sum of X divided by A and W times 598.

```
LET YY = X/A + W*598
```

Note here that no parentheses are needed because the normal order of precedence is to be followed.

Table 4.1 Evaluation of Logical Expressions

This Section describes the logical or relational operations (words and signs) that the ATARI Personal Computer can use.

Operator Sign or Symbol	Definition	Example of Use
=	Is equal To	A = B (A is equal to B)
<	Is Less Than	A < B (A is less than Ø)
<=	Is Less Than or Equal To	A < = B (A is less than or equal to B)
>	Is Greater Than	A > B (A is greater than B)
>=	Is Greater Than or Equal To	A > = B (A is greater than or equal to B)
<>	Is Not Equal To	A < > B (A is not equal to B)
NOT	Logical Negative	NOT A (Statement is true if A is false)
AND	Logical Both	A AND B (Statement is true only if **both** A AND B are true)
OR	Logical Or	A OR B (Statement is true if **either** A OR B or both are true)

Order of Precedence: Same as arithmetical operators, followed in order of precedence by **NOT**, **AND**, and **OR**.

Parentheses: Parentheses are used to change the normal order of execution priority, just as with arithmetic expressions. "Nesting" parentheses is perfectly acceptable.

CHAPTER 5

THE INS AND OUTS OF IT ALL: I/O OPERATIONS

Ladies and Gentlemen! If we can get back on the bus, we have a long way to go today! All aboard, please, for our ATARI tour of peripheral devices, and all the ways we have to get them to cooperate with us! Let's go!

Input/Output (or just I/O) operations are one of the least understood and most important phases of computer science. We can have the fastest, smartest system in the world (with megabytes of memory storage, chrome wire wheels, air conditioning and steel-belted radial tires) but if we can't get a program **in** and the answers **out**, it's about as useful as gills on a butterfly!

I/O operations are often misunderstood, and the subject is avoided when possible. This is **not** because it is difficult, but because there are often so many control words and convenience features that most systems are less than perfectly logical in their approach to the most important single part of the system: **you**! The human operator!

The ATARI Personal Computer System is relatively simple, internally consistent and logical, and the keywords are generally applicable, regardless of the particular I/O device being used.

5.1 The I/O Peripherals

It seems in order at this point to take a very short side-tour of the devices that can be used with your ATARI Personal Computer System. The hardware-side is well covered in the individual manuals furnished with each device, so we will concentrate here on the programmer's viewpoint. Figure 5.1 shows a "full-blown" ATARI 800 system. Let's briefly describe each piece of equipment (hardware) in the system:

Keyboard: This is the only "pure" input device available at present for getting information into the computer. By "pure" we mean as opposed to a device that can conduct both input **and** output operations. But you can't get information **out** of the computer through the keyboard!

Important Note: When discussing I/O operations and peripheral devices, bear in mind that the words **Input** and **Output** are used **in relation to the computer.** We are always speaking of getting data into or out of the central processing unit (CPU or MPU).

Line Printer: The ATARI 820™ and ATARI 825™ Printers are the only "pure" output devices. (Even the TV screen is not thought of this way, as we shall see). There is no way a printer can send information into the computer! It is for output only. (For you folks with printer experience, the ATARI 820 and ATARI 825 Printers are dot matrix impact-type printers.)

Program Recorder: The ATARI 410™ cassette Program Recorder (which will work with either ATARI 400 or ATARI 800 Personal Computer Systems)
is **both** an input and an output device! Programs or data can be recorded from RAM onto cassette tape, and then later can be reloaded into RAM from the tape. This is a dandy way to save programs so they don't have to be constantly retyped into the keyboard each time you want to **RUN** them. Actually, this is a custom cassette tape deck that uses two tracks for sound and program recording purposes.

Disk Drives: From one to four ATARI 810™ Disk Drive units may be used with systems with 16K RAM or more installed. These units perform functions similar to those performed by the Program Recorder, but they do it in seconds instead of minutes. (See your ATARI dealer for more information on these extremely rapid data and program file storage devices.)

Screen Editor: A type of direct memory access (DMA) is used in the ATARI Personal Computer Systems, and the editor memory is considered "just another" I/O device. In the split-screen graphics modes (see Chaper 9), the editor I/O device handles the four text lines at the bottom of the screen. We must train ourselves to think of this editor as a typical I/O device capable of normal I/O operations, and not "just" a part of the computer, even though the device "lives" in the computer console and no additional hardware is required.

TV Monitor: Uses DMA also, so the computer can consult RAM at any time to "read" what is displayed on the screen. In this sense, at least, an input operation is possible, although we humans will probably persist in thinking of the TV screen as an output device. Those of you with some experience in the ways of computers and peripherals will understand that when we say that "memory-mapped I/O is used", we can and should think of the monitor screen as an input device as well as a pure display of information. Note that in the graphics modes with split-screen, the screen I/O device (S:) is used with the graphics (larger, top portion of the screen) while the Editor device (E:) handles the text window. (Again, please see Chapter 9 for a complete description of the graphics modes. Also see Appendix F, covering the keyboard and editor.)

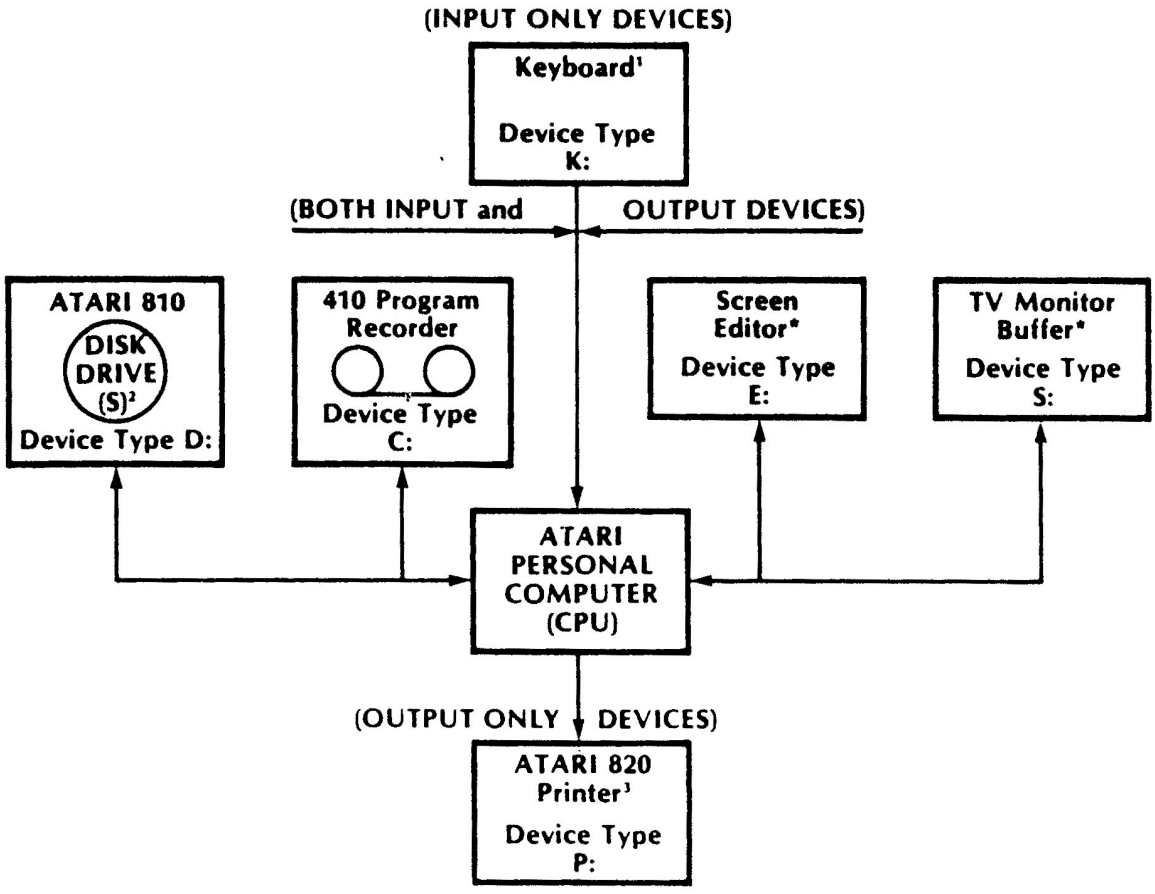
You can bet that more I/O or "peripheral" devices will be forthcoming for your ATARI Personal Computer System! And that's the beauty of the "system approach". These will be easy for you to use and **understand** if you will take the time to develop a "picture" of I/O operations in general.

Figure 5.1 is a graphic representation of the various currently available I/O peripherals you can use with your personal computer. Note that you already have (even if you have the ATARI 400) the Editor, Screen, and Keyboard devices!

Figure 5.1:

INPUT/OUTPUT (I/O) OPERATIONS

Fully Developed ATARI System



1 "Hardware" is part of Computer Console.
2 Up to 4 Disk Drives may be used with the ATARI 800 Personal Computer System.
3 Only one Printer should be connected.

5.2 Software Control of All That Hardware:

When we consider computer control of six or more peripheral devices, we have to wonder how the central processing unit (CPU) can control them all. It sounds very complicated but really isn't! Let's try to put ourselves "in the shoes" of that tiny lump of silicon that has to do the job, and try to ascertain just what it needs to "know" to do this control job. (Just how "smart" do you have to be to "boss" six or more I/O operations at once, and even more importantly, just **what** do you have to know?)

1. Well, if I were a CPU, first I'd want to know what kind of device will do the job my master wants done.
2. Oh! A disk drive, huh? Which one shall I use? (I am a lucky CPU: I get to boss four disk drives.)
3. How can I tell if this is to be an **input** or an **output** operation?
4. Is it really an I/O operation at all? Maybe the boss just wants me to generate all those income tax figures for my own information.
5. Gotcha, Boss! You want me to "write" all that onto disk #3. OK, but I'll need to know the file name so I can get it in the right place on the disk.

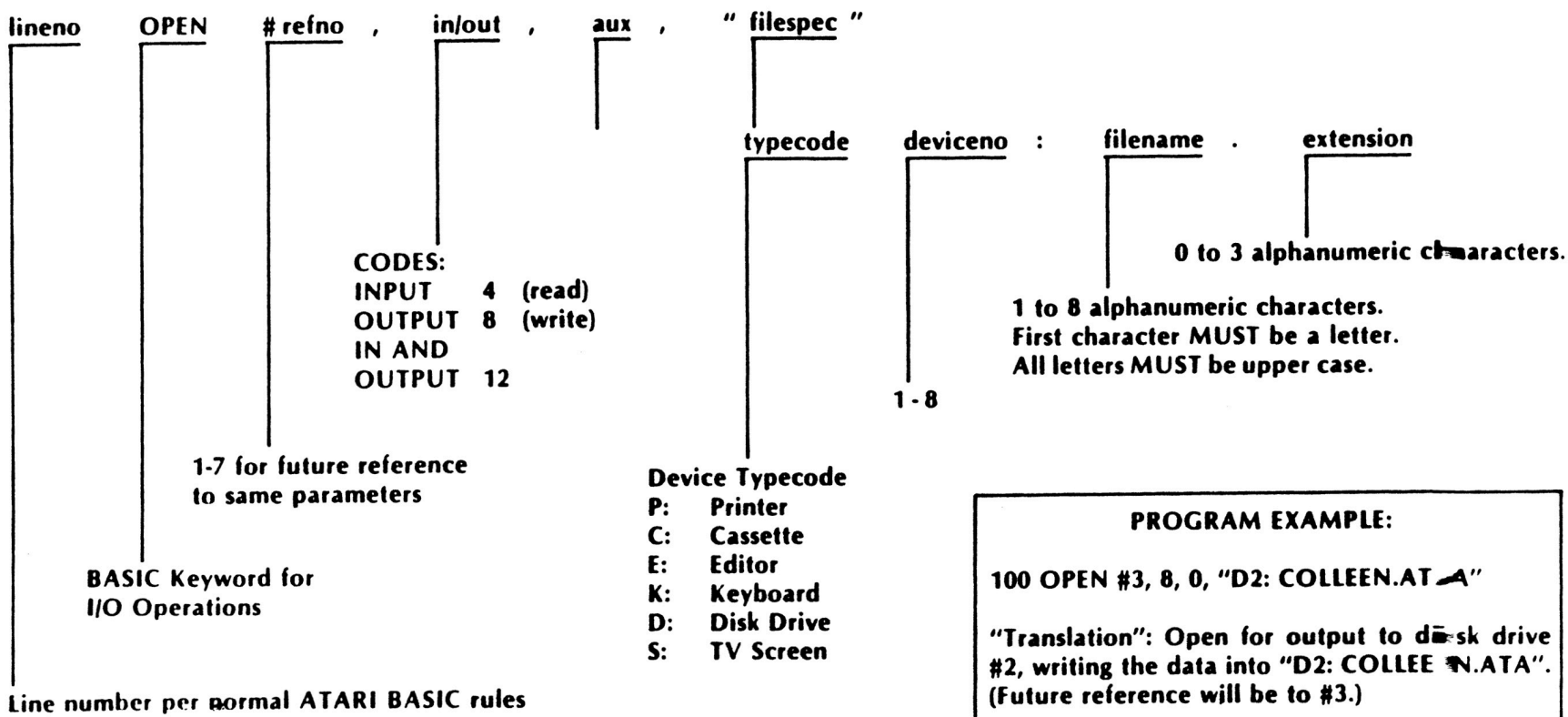
We can summarize this dialogue with a CPU as follows:

1. Is this to be an I/O operation?
2. **Input** or **Output**?
3. Device type to use?
4. Which device (if more than one of the same type)?
5. File to get info from or store it in? File name?

(See Figure 5.2)

NOTE: For purposes of clarity, we have temporarily abandoned the conventions described in chapter 1. The reasons for this will become obvious, for the OPEN statement is a very general Input/Output statement and has many options and may take many apparently different forms.

Figure 5.2 - THE OPEN STATEMENT - GENERAL FORMAT



NOTE also that "filespec" may be expressed as a string variable: 100 A$ = "D2: COLLEEN. ATA" 110 OPEN #3, 8, 0, A$

PROGRAM EXAMPLE:

100 OPEN #3, 8, 0, "D2: COLLEEN.ATA"

"Translation": Open for output to disk drive #2, writing the data into "D2: COLLEEN.ATA". (Future reference will be to #3.)

NOTE that all punctuation, the # symbol, and the quotation marks are REQUIRED!

5.3 The OPEN Statement

The **OPEN** statement is the BASIC key to the whole I/O system. Once it has been thoroughly absorbed and understood, and you see what can be done with it, it will open the door to the other I/O concepts we will cover later. For this reason, we will devote an entire section to **OPEN**.

OPEN

Format: lineno OPEN # aexp, aexpl, aexp2, "filespec"
Note that "filespec" may be a string variable!

Abbreviation: O.

Program Example:

```
100 OPEN #2, 8, 0, "D1:ATARI800.BAS"
or
100 A$ = "D1:ATARI800.BAS"
110 OPEN #2, 8, 0, A$
```

Notes On OPEN

1. **lineno** is the normal ATARI BASIC line number.
2. **OPEN** is the BASIC keyword
3. **#** is a **mandatory** character which **must** be typed in.
4. **aexp** is an expression that evaluates to the number that will be used later to refer to the same set of parameters.
5. **aexpl** may evaluate to 4, 8, or 12. If 4, an input (to computer) operation will take place, if 8, an output operation, and if 12, both input and output may occur
6. **aexp2** is an auxiliary printer control code. If no auxiliary code is desired, a 0 (zero) must be entered.
7. **filespec** is a specific file designation. The quotation marks around **filespec** are required unless a string variable is used
8. The commas (see format and program examples above) must also be typed into the computer
9. **filespec**, if used, consists of the device code (see Figure 5.2) followed by a number between 1 and 8 followed by a required colon (:) followed in turn by the filename and extender. Example: D1:FILENAME.EXT
10. The **filename** may contain **up to 8** alphanumeric characters and **must** start with an alphabetic character (a capital letter). For example, "FILEONE", "ZIPZAP", "A1111111", and "DISKFILE" are all legal filenames. **No lower case or non-alphanumeric characters may be used.**

11. **Optionally, a file extender** may be "tacked on" to the end of a filename by placing a period **immediately** (no spaces) after the filename and then adding the extender **immediately** after the filename and period. The **file extender** may consist of any three (or less) alphanumeric characters. For example "MAUREEN9.339" is legal, as are "KENT921.61" and "KAREN711.63".

All this is mostly by way of introducing the concept of an I/O system that controls all connected devices using the same BASIC keywords and format. Here's a list of the device type codes used:

FIGURE 5.2 DEVICE CODES

Code	Device
D	Disk Drive
C	Cassette (Program Recorder)
K	Keyboard
E	Editor
S (default device)	Monitor (Screen)
P	Printer

See 9 above for use of the device codes. Note that these are actually codes that tell the computer what **kind** of device is to be used, not which of several disks, for example, is to be used.

Program Examples:

100 OPEN #3, 4, "D1:JEAN.W"

Translation: Open for input disk file "D1:JEAN.W" and assign to the reference number that the expression #3 evaluates to (3, of course!). The whole set of parameters can now be referred to by the number #3. more on this later!

100 OPEN #1, 8, "D1:CHEKBOOK.BAL"

Translation: Open for output to disk no. 1 the file that will be referred to in the future simply as #1, and which is titled "D1:CHEKBOOK.BAL" on the disk.

Once again, remember that **things are not really this complicated,** for the "full" **OPEN** statement protocol is seldom if ever used other than with disks. There are short-cuts and other BASIC I/O keywords yet to be covered that will simplify matters tremendously. **It is very important, however, that you "latch on" to the concept of centralized input/output operations with a system that is consistent, even though there are many shortcuts that work with specific devices.**

Let's sum-up our knowledge of the **OPEN** statement by saying that it is used to establish a reference number so we don't have to go through the whole routine more that once, tells the computer whether it will be an input or an output operation, establishes what type of device will be used, controls which of several devices (of the same type) will be controlled, and tells the computer what the filespec (file specification) is on the particular disk. **Note:** Filenames are not used with the program recorder.

We should also realize at this point that **more than one** I/O operation may be **OPEN at any given time.**

CLOSE

Format: lineno CLOSE # aexp
Abbreviation: CL
Program Example: **100 CLOSE #2**

This simply **CLOSE**s files that have been previously **OPEN**ed. Note that the number to which **aexp** evaluates will be the same number as that of the **aexp** following the # symbol in the **OPEN** statement that the programmer now wishes to **CLOSE**. Here's an example:

```
10 OPEN #4, 4, 0, "D1:XXXXXXXX.YYY"     (OPEN the file for
                                          read
                                          (input) operations)
20                                        (I/O statements)
30                                        (I/O statements)
40 CLOSE #4
```

Now you see what that first aexp (the one following the #) was really for! It saves us having to go through the whole routine for **CLOSE** that we did for **OPEN**. Well, things will get even easier, Folks!

It should be noted at this point that **NEW**, **END**, and **RUN** perform the **CLOSE** operation automatically, and that once a file has been **CLOSE**d, the **OPEN** statement is "forgotten" by the computer. If the same file must be re-**OPEN**ed, this must be done in the same way in which it was **OPEN**ed originally. Of course a different reference number may be used if desired.

Try this little program on your system. It will be both fun and educational!

```
NEW                                  (don't forget this when writing a new
                                      program)
READY                                (says the trusty computer)
10 OPEN #1, 4, 0, "K:"               (Open the keyboard device for input)
20 GR.0                              (Clear screen for action!)
30 PRINT " ": GET#1, X               (Input the byte of the key that was
                                      pressed) (More on GET later)
40 FOR N= 1 TO 1000: NEXT N          (Time delay loop)
50 PRINT X                           (Print the key pressed)
60 PRINT CHR$(X)
70 FOR N= 1 TO 1000: NEXT N          (Another delay loop)
80 GOTO 20                           (Go back and do it again)
```

RUN this program in the normal way. The cursor will appear. Now press one letter or other symbol on the keyboard, **but do not press** RETURN . Watch the screen. About two seconds after you press any key, it will be printed along with its numerical ATASCII code) on the screen. Now you can press another key and repeat the process. Note that you are entering letters one at a time into the computer without pressing the RETURN key, without using string variables, and without automatically printing the letters on the screen as you enter them! A new way to exercise your skill as the "master". You should be able to think of all kinds of uses for this! Bear in mind that it is the I/O capabilities of your system that make all this possible.

One more point on the **OPEN** statement and its shadow the **CLOSE** statement. The **GRAPHICS** statement (see Chapter 9 covering ATARI Graphics) automatically **OPEN**s the screen (S:) as reference #6.

5.4 The PRINT Statement: Another Generalized I/O Control

PRINT

Format: lineno PRINT #aexp ; exp ; exp... ;
Abbreviation: ?

Program Examples:

```
100 PRINT "THE VALUE OF X IS "; X
100 PRINT "COMMAS", "CAUSE", "COLUMN", "SPACING"
100 PRINT X, Y, Z, A$
```

To simplify just a bit, we can say that a **PRINT** statement is like a command to "put" something somewhere. If we do a **PRINT** to a printer, we get hard copy. If we **PRINT** to the screen we get a picture on the TV set. Normally we use **PRINT** only for those two things, but not always! The aexp following the # in the statement is the (optional) file specifier between 1 and 7 that controls to which device the **PRINT** will be done. If no aexp is used here, the "default" mode will choose the TV screen. We should note at this point that either a comma or a semicolon may be used separating two **PRINT**ed items, and that quotation marks may be used, too.

Use of Punctuation: A comma following anything to be **PRINT**ed causes tabbing to the next tab location before **PRINT**ing resumes (even if the next item to be printed is on the following numbered line). In contrast, the semicolon following something **PRINT**ed (or separating two **PRINT**ed items) causes the succeeding printed items to be "jammed" against the preceding one (again, even if it is on another line number!) If no punctuation mark follows the last **PRINT**ed item, the next **PRINT** will start on the following line.

Quotation Marks: Quotation marks around any item to be **PRINT**ed will NOT themselves be **PRINT**ed. They mean to the computer that it is to **PRINT**, character for character, exactly what is between the two quotation marks.

Thus:

```
NEW
10 X=1: Y=2: Z=3
20 PRINT "X";"Y";"Z"
RUN
XYZ
READY
```

If we remove the quotation marks, we get this:

```
NEW
10 X=1: Y=2: Z=3
20 PRINT X;Y;Z
RUN
123
READY
```

We can change line 20 to read like this: 20 PRINT X; " ";Y;" ";Z. Now, the blank space we put in the quotation marks will be printed, and this will leave blanks between the numbers:

```
RUN
1 2 3
READY
```

Or, we can use commas instead of the semicolons and remove the quotation marks.

```
20 PRINT X,Y,Z
RUN
1       2       3
READY
```

5.5 "Shortcut" Controls for Specific Devices

We have now discussed the **OPEN** and **CLOSE** statements, and have learned that **PRINT** is another generalized I/O statement. We have seen that there is a pattern in all this! We can choose any I/O device with an **OPEN** statement, assign it a reference number, get information in or out of the device, **CLOSE** the I/O Operation, and then **OPEN** another. (We can have several reference numbers **OPEN** at the same time, too.)

But that sounds like an awful lot to remember just to print out one line on our printer, record a program on tape, or load a program into RAM from our disk drive. Isn't there a quicker way?

Naturally! You bought a first class computer when you bought your ATARI Personal Computer System! Let's look at the various devices you will be making the most use of, and discuss the statements you'll need to control them "the easy way":

LPRINT

Format: **lineno LPRINT exp [{;} exp {;} ...]**

Abbreviation: **LP.**

Program Example:

```
LPRINT "THE QUICK BROWN FOX JUMPED OVER
THE LAZY DOG"

100 LPRINT X;" ";Y;" ";Z
```

This statement causes the computer to output to the printer rather than the screen. The **LPRINT** statement requires no **OPEN** or **CLOSE** statements and no file specifier. Just tell it what you want it to **LPRINT**.

In case you were wondering, the **L** in **LPRINT** stands for "line". We are dealing here with a line printer (as opposed to a screen printer).

But how do I get a program **LIST** out of it for my notebook? Easy again! Just tell it (in direct mode) to **LIST"P:** It will now faithfully "grind out" a complete listing of your latest brain-child. Of course **LIST"P:** can be used in deferred mode (if you just **have** to) by adding a line number. We'll come back to **LIST** later, and discuss it in conjunction with its "sister" statement, **ENTER**.

TABLE 10.1
TABLE OF PITCH VALUES FOR THE MUSICAL SCALE

HIGH NOTES	C	29
	B	31
	A#	33
	A	35
	G#	37
	G	40
	F#	42
	F	45
	E	47
	D#	50
	D	53
	C#	57
	C	60
	B	64
	A#	68
	A	72
	G#	76
	G	81
	F#	85
	F	91
	E	96
	D#	102
	D	108
	C#	114
MIDDLE C	C	121
	B	128
	A#	136
	A	144
	G#	153
	G	162
	F#	173
	F	182
LOW NOTES	E	193
	D#	204
	D	217
	C#	230
	C	243

"C" SCALE MUSIC PROGRAM

This little program will demonstrate some of your computer's musical talents, and will show the use of the **READ...DATA** statements in musical programming:

```
10 READ A                          (read the data for variable A)
20 IF A = 256 THEN END
30 SOUND 0, A, 10, 10              (1st voice, pitch A, normal tone,
                                    slightly louder than normal volume)
40 FOR W = 1 TO 400: NEXT W
50 PRINT A                         (print the value of pitch being played)
60 GOTO 10                         (loop back for next note)
70 DATA 29,31,35,40,45,47,53,60,   (data to be read in line 10)
   64,72,81,91,96,108,121
80 DATA 128,144,162,182,193,
217,243,256                        (256 is end of data marker)
```

Note that this program will end when the last data statement (256) is **READ** by line 10 and line 20 is executed. Also note that all **DATA** statements are at the end of the program. Actually, **DATA** statement can be placed anywhere in the program.

10.3 And Now, the Game Controller Functions

This may come as a surprise to some of you: Not only can you "plug in" ATARI game cartridges and other programs that you can operate using the keyboard, paddle controllers, and joysticks, but you can use these controls to **RUN** programs for games (or whatever) that **you write** in BASIC! How's that for a nice way to end our evening out and wind up our tour of BASICland?

ATARI calls the control knob or wheel a "paddle" controller (because it was originally used to control the paddles displayed in the PONG® games. We at ATARI still think of it as the "paddle" controller, so that's what we'll call it in this book.) The following two functions (yes, they are true functions, as values are "returned") are for use with the paddle controller devices:

PADDLE

Format: **PADDLE (aexp)**
Abbreviation: **None**
Program Example: **100 PRINT PADDLE(3)**

This function returns the status of a particular numbered controller (The controllers are numbered 0-7 from left to right). This can be used in conjunction with other functions and statements to cause "things to happen" like sound, graphics controls, etc. For example, **IF PADDLE(3)=14 THEN PRINT "PADDLE ACTIVE!"** Note that the number returned by the **PADDLE** function will be between 1 and 228, with the number increasing in size as the knob on the controller is rotated counterclockwise.

PTRIG (Paddle Trigger Function)

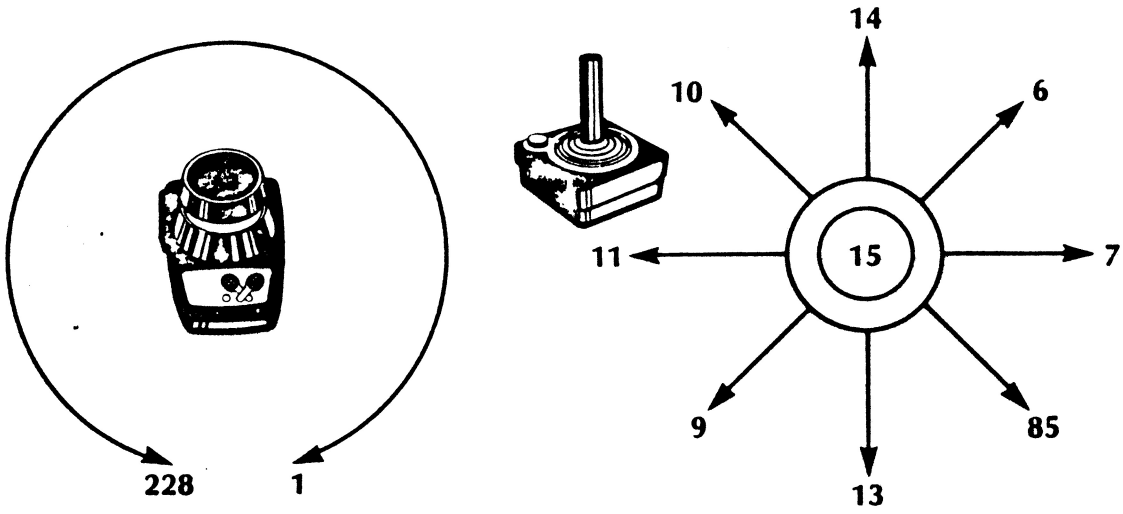
Format: **PTRIG (aexp)**
Abbreviation: **None**
Program Example: **100 IF PTRIG(4) = 0 THEN PRINT "MISSILES FIRED!"**

(aexp must evaluate to a value between 0 and 7)

PTRIG returns a number that represents the status of the trigger button on the particular paddle controller. "1" is returned at all times unless the button is pressed. In that case, **PTRIG** returns a "0".

STICK and **STRIG** are to the joystick controller what **PADDLE** and **PTRIG** are to the paddle controller, and they work exactly the same way. Figure 10.1 shows the numbers that will be returned when the joystick is pushed in any direction. Note that like **PADDLE** and **PTRIG, STICK** and **STRIG** have no abbreviations.

FIGURE 10.1 PADDLE, STICK, PTRIG, and STRIG



Note: **STRIG** and **PTRIG** both return a "1" until trigger button is pressed, at which time "0" is returned.

The imaginative programmer will think of many uses for the four functions described in this section, as they are the easiest and most direct way to access the computer **while a program is actually running**. These functions may be used to produce musical notes, graphic effects, and the like. The controllers may even be connected to external mechanical devices so that outside events can be fed directly to the computer for processing and control purposes. These unusual input features may be used in any way you like, and the only real limitation is your own human imagination! In fact, the answer to most questions about computers is "Yes, it can do that..." The limiting factor is **nearly always** between the programmer's ears!

Hope you enjoyed tonight's special tour! Sound and games are one of the most interesting and entertaining areas of the computer world. Tomorrow we'll end our tour of BASICland with a few advanced programming concepts and "hints & kinks" that will help you tie everything you've learned about the ATARI personal computers together a bit better. See you in the morning!

CHAPTER 11

ADVANCED PROGRAMMING TECHNIQUES IN BASIC

Well, folks, that about does it! You've had your tour of ATARI BASICland, with all the stops along the way. That means we only need to spend one more day to help you tie it all together, introduce a few more advanced statements and functions, talk a little about memory conservation, and pass on some tips for good programming.

11.1 PEEK, POKE, and USR

We've mentioned these BASIC keywords during other parts of our tour, but we've been saving their full use for the end. This is **not** because they are any harder to master than the rest, but because there are certain restrictions or limitations involved with their use. For example, **USR** is a function that "calls" program resident in **RAM** in **machine code. POKE** allows us to directly alter the contents of a particular memory location, and this exposes us to potential damage to a long program that is in **RAM. PEEK** is really a companion of **POKE**, and the two belong together. Let's take 'em as they come:

PEEK

Format: **PEEK (aexp)**
Program Example: **100 PRINT "LEFT MARGIN IS "; PEEK(82)**

This function allows the user to **PEEK** into a particular memory location and "see" the byte stored there. In the above program example, the number returned will be the setting for the left margin of the screen display. This number "defaults" to 2, so if 36 is returned, the programmer would know that this margin has been changed for some reason. Any memory address (in decimal form) in ROM or RAM may be **PEEK**ed without changing the data stored in it. So **PEEK** away (usually in direct mode) to your heart's content!

Note that **both** aexp and the number returned by **PEEK** are in **decimal** (not hex) form!

POKE

Format: **lineno POKE aexp1, aexp2**
Program Example: **10Ø POKE 82, 1Ø**

This is really the opposite of **PEEK**. Instead of a "passive" **PEEK** into a memory location (aexp1) to see what is there, this is an aggressive **insertion** or modification of data stored in the memory location. In the format example above, aexp1 is the address (again, decimal form) of the location. Note that aexp2 is also in decimal format and is between 0 and 255. If you try the program example given above (do it in direct mode, without a line number), you will see that **POKE**ing memory location 82 with the number 10 will change the left screen margin from its default (2) position to 10; that is, the margin will move 8 spaces to the right. (Hit SYSTEM RESET to restore the margin to normal.)

Like other BASIC keywords, **PEEK** and **POKE** can be used in direct as well as deferred mode. Remember that performing a **POKE** actually alters a byte stored in RAM (you can't **POKE** ROM locations, so don't try!), so if the wrong location is **POKE**d, or if the wrong byte is **POKE**d into the right location, the computer is likely to respond by saying "Ouch!" in its own inimitable fashion by doing strange things to your precious program! Indiscriminate **POKE**ing is to be avoided for this reason. You may well find yourself with a machine that is "hung up" or with a "crashed" BASIC system. In these cases, you will be unable to communicate with the computer at all; an "offended" computer will just ignore everything you do! Try first pressing SYSTEM RESET. If this works, your program will still be in RAM and can be **LIST**ed or **RUN** again. If all else fails, power down by turning the computer off, wait about 5 seconds, and power up again. This will re-establish contact between you and the machine, but your program (and anything else stored in RAM) will, or course, be lost!

REMEMBER:
Do your experiments with POKE when you do not have valuable data stored in RAM! It is impossible to hurt the computer's hardware by anything you may do at the keyboard, but data and programs can easily be lost if you "POKE" around without knowing what you are doing!

In practical programming, you will find that **ATARI BASIC** is a very flexible and versatile programming language, and it is good practice to resort to **POKE**ing **only** when dealing with very special considerations that are beyond the capabilities of BASIC!

One last tip when preparing to use the **POKE** command: **PEEK** first! Jot down the byte that **was** in the particular location. Then, if the **POKE** doesn't work out as you anticipated, you can **POKE** the original byte back in there where it "belongs!"

USR (Function "Calls" a User 6502 machine-language subroutine and returns the result of its execution)

Format: **USR (aexp1 [, aexp2 [, aexp3...]])**
Abbreviation: **None**

Notes: aexp1 is an integer or arithmetic expression that evaluates to an integer which represents the decimal memory address of the machine language routine to be performed.

aexp2, aexp3 etc. are the **optional** input arguments (an argument is really nothing more than the value of a variable that is passed to the subroutine, so don't let the new terminology scare you) for the subroutine. These should be arithmetic expressions which evaluate to an integer between 0 and 65535 inclusive, but a non-integer value may be used. (It will be rounded to the **nearest** integer.) These values will be converted from BASIC's BCD floating-point number format to a two-byte binary number, and then they will be pushed onto what is called the "hardware stack". This stack is a group of RAM memory locations under the direct control of the 6502 microprocessor chip, and we can think of the stack as looking like Figure 11.1 below.

FIGURE 11.1 THE HARDWARE STACK

(TOP OF STACK)

N (Number of arguments on the stack, which may be zero)
X_1 (High byte of argument X)
X_2 (Low byte of argument X)
Y_1 (High byte of argument Y)
Y_2 (Low byte of argument Y)
Z_1 (High byte of argument Z)
Z_2 (Low byte of argument Z)
:
R_1 (High byte of RETURN address)
R_2 (Low byte of RETURN address)

Note: X is the first argument following the address of the routine, Y is the second, Z the third, and so on. There are N **pairs** of bytes.

The **ADR** function (see Chapter 6) may be used to pass data that is stored in arrays or strings to a subroutine in machine language. Use the **ADR** function to get the address of the array or string, and then use this address as one of the **USR** input arguments. See Appendix I for a much easier BASIC way to do this!

Before it returns to BASIC, the assembly language routine must **POP** the number of input arguments (N) (top item on stack) off the stack. If this number is not 0, then all of the input arguments must be **POP**ped off the stack also. See Figure 11.1.

The subroutine should end by placing the low byte of its result in location 212 decimal in RAM and the high byte in location 213 decimal, and then return to BASIC using an **RTS** (return from subroutine) instruction. The BASIC interpreter will convert the 2-byte binary number stored in locations 212 and 213 into an integer between 0 and 65535 in floating-point format to obtain the value returned by the **USR** function.

One way to get an assembly-language routine into memory is to read it into an array, one byte at a time. First, the routine must be assembled either by hand or with a 6502 assembler program, such as ATARI's Assembler Editor cartridge[1] which simply plugs into your ATARI Personal Computer System console. This program-cartridge will provide the hexadecimal values required in your machine-level subroutine. These are next converted to decimal and placed in a **DATA** statement. The address of the array is then obtained by using a **DIM** statement with a string **DIM**ensioned to length 1 followed by the array **DIM**. The address of the array is obtained by adding 1 to the address of the string...like this:

```
100 DIM A$(1), A(10)
110 PRINT ADR(A$) + 1
```

The object file (machine code) from the Assembler cartridge (or whatever) may be stored on disk or cassette, then entered into memory either with the use of **DOS** (Disk Operating System) or with the **GET** statement in BASIC. See the manual furnished with the Assembler-Editor cartridge for the format of the object file. (See also Appendix I)

See Appendix D for Decimal vs. Hexadecimal conversion tables. "Hexadecimal" is computerese for the base 16 numbering system.

[1] Available late 1980

USR, like **POKE**, involves a certain amount of knowledge of machine language programming and should not be attempted without some advanced study. We recommend the purchase of the ATARI Assembler Editor cartridge and careful study of ATARI's publication for newcomers to the strange world of machine language programming.

Like **POKE**, the **USR** function can be mistreated and may cause your computer to go off by itself and sulk (and not even speak to you!) Try SYSTEM RESET , and if this fails, turn it off, wait about 5 seconds, and power up again. (Sure hope you had that program on tape before you started fooling around with **USR** and **POKE**...) (See Appendix I for more on **USR** and executable program examples)

11.2 On The Conservation of Memory

Your RAM memory (no matter how much you have) will always be just barely enough for the program you want to write, provided you are very careful, even stingy at times. This is one of those "laws" like the one about how a part that can be installed backwards **will** be so installed. Computer or human, **none** of us **ever** has enough memory! The world of the personal computer is often **quite** limited in this regard, since RAM **is** expensive, so it behooves us to know our equipment and the techniques we can use to conserve valuable RAM without sacrificing the human element in our programs. The latter is **very** important, as one of the major differences between your ATARI computer and a hand-held calculator is the computer's ability to **simulate** "human" thinking. Unless memory is severely limited, we should take a humanistic approach to programming. So let's learn how to save memory where doing so "won't show".

1. In many small computers, eliminating blank spaces between words and characters as they are typed into the keyboard will save memory. This is **not true** of the ATARI Personal Computer Systems. This means you can write as if you were using a typewriter. This will prove enlightening both to humans who must work with your programs **and** to your computer!

   Spaces should be used (just as in typing on a conventional typewriter) between successive keywords and between keywords and variable names. Here is an example:

```
10 IF A=5 THEN PRINT A
```

   Note the spaces between IF and A and between THEN and PRINT. In most cases, a statement will be interpreted correctly by the computer even if all spaces are left

out, but this is **not always** true. Almost as important, you will find it much easier to work with your program if you use conventional spacing. Note, too, that when you give the command **LIST**, the computer will put in its own spacing, regardless of how many spaces you typed in originally. This is a very **smart** computer! If you don't believe this, try typing in a line leaving extra spaces (use deferred mode) and then tell your electronic friend to **LIST**!

You can easily confuse the poor thing if you put spaces in the middle of variable names, keywords, or numbers. Do **not** put a space between an array variable name and the left parenthesis that follows it!

Do it This Way:	**Never This Way:**
A(0)	A (0)

2. Each new line number represents the beginning of what is called a new "logical line". Each logical line eats up 6 bytes of "overhead", whether it is used to full capacity or not. Adding an additional BASIC statement by using a colon (:) to separate each pair of statements on the same line "costs" only 3 bytes.

If You Need To Save Memory,

AVOID PROGRAMS LIKE THIS:

```
10 X = X + 1
20 Y = Y + 1
30 Z = X + Y
40 PRINT Z
50 GOTO 10
```

CONSOLIDATE LINES LIKE THIS:

```
10 X = X + 1: Y = Y + 1: Z = X + Y: PR
INT Z:
GOTO 10
```

(NOTE: 12 bytes were saved)

If memory is not a problem, then you may want to put most statements on separate numbered lines in the interests of programmer convenience and readability.

3. Variables and constants should be "managed" for savings, too. Each **time** a constant (4, 5, 16, 3.14159, etc.) is used, it "costs" 7 bytes. Defining a new variable requires 8 bytes plus the length of the variable name (in characters). But each time it is used after being defined, it takes only 1 byte, **regardless of its length!** Thus, if a constant (such as 3.14159) is used more than once or twice in a program, it should be defined as a variable, and the variable name used throughout the program. For example:

```
10 PI = 3.14159
20 PRINT "AREA OF A CIRCLE IS THE RADI
US SQUARED TIMES "; PI
RUN
AREA OF A CIRCLE IS THE RADIUS SQUARED
TIMES 3.14159
READY
```

4. Strings require 2 bytes overhead and 1 byte for each character (including all spaces) in the string. (See string variables below.)

5. String variables cost 9 bytes each plus the length of the variable name (including spaces) plus the space eaten up by the **DIM** statement, plus the size of the string itself (1 byte per character, including spaces) when they are defined. Obviously, the use of string variables is very costly in terms of RAM.

6. Arrays and matrices are also memory burners, but in many cases are good investments from other standpoints. (Much of our computing power comes from array/matrix operations.) Definition of a new matrix requires 15 bytes plus the length of the matrix variable name plus the space needed for the **DIM** statement plus 6 times the size of the matrix (product of the number of rows and the number of columns). Thus, a 25 row by 4 column matrix would require 15 + approximately 3 (for variable name) + approximately 10 (for the **DIM** statement) + 6 times 100 (the matrix size), or about 630 bytes!

7. Many programmers use a **lot** of **REM** statements during the creation of a new program. For both memory conservation **and** increasing execution speed, these may be removed once the program is "**RUN**ing" properly. Remember, though, that your remarks will help others to understand your program.

8. Many times, certain operations must be repeated on different variables or constants, timing loops are used, and other oft-used sections of the status, and called with **GOSUB** when needed. Storing each of these routines only once in the program will save quite a bit of memory, not to mention your own typing time!

Unnecessary use of parentheses costs you one byte per character. But be sure you know where you can safely remove these before you start doing "parenthessectomies" on your programs! Review Chapter 4.

The above information and suggestions are included for the information of the reader, for it is not ATARI's intent to create a sub-race of byte-obsessed compulsive memory counters! We do, of course, realize that many computer hobbyists are interested in such things, and since all memory has limits, it makes common sense to be aware of ways to conserve it.

11.3 Good Practices and Hints for Better Programming

As usual, experience is the best teacher. However, there are many traps and pitfalls out there in BASICland that are due to the literal nature of computers. If you could command (in BASIC) your computer to "JUMP", it would almost surely present you with an error message to the effect that you haven't told it how high! Here are some good programming practices and helpful "dos and don'ts" that will help you take your computing fun as seriously as we at ATARI do.

1. **Never** use variables that look like BASIC keywords like **RUN, POKE, GOTO**, and the like. This will eventually confuse both the programmer **and** the poor computer!

2. Be aware of the order of arithmetic and logical execution, and make judicious use of parentheses ( ). If you have trouble, refer back to Chapter 4. And if in doubt, use parentheses, even though they "cost" a byte each. Chances are you have memory "to burn" anyhow!

11.4 PEEKs, POKEs, Pulses and Programs

Note: Many of these locations are of primary interest to *expert* programmers and are included here as a convenience.

This section contains a few more hints and bits and pieces of information that may prove useful to your programming efforts:

1. **Timing Operations:** You can obtain clock pulses derived from the crystal controlled oscillator that serves to time your computer's internal workings. These are stored in memory locations 18, 19, and 20 (decimal). Location 20 changes with each TV monitor frame, or 60 times per second. Location 19 changes every 4.27 seconds, and 18 changes every 65536 TV frames, or 18.2 minutes. Use of the algorhythm shown below as line 100 in a hypothetical program will get you a count that increments by one with each second that passes:

```
100 SECONDS = INT((PEEK(18) * 65536+PEEK(19) * 256+PEEK(20))/60)
```

2. **Controlling the Program Recorder Motor:** The statement **POKE 54018,52** will turn on the Cassette Recorder motor, while **POKE 54018,60** returns the motor to its normal off state. This may be used in direct mode or in a program to play tape recorded music during a program or for any other purpose. Music or data on cassette will be played through the TV set's loudspeaker, and may be controlled by the TV's volume control.

TABLE OF POTENTIALLY USEFUL MEMORY LOCATIONS

DECIMAL LOCATION	COMMENTS AND DESCRIPTION
14, 15	Highest location used by BASIC (LSB, MSB)
18,19,20	TV frame counter (1/60 sec.) (LSB, NSB, MSB) see above
65	Noisy I/O Flag (O = quiet)
77	Attract Mode Flag (128 = Attract Mode)
82,83	Left, Right Margin (Defaults 2, 39)
84	Current row on which cursor rests (0-23)
85, 86	Current cursor column (0-39)
90	Previous cursor row (0-23)
91, 92	Previous cursor vert. column (0-39)
93	Data under cursor
96	Cursor row to which DRAWTO will go
97,98	Cursor column to which DRAWTO goes
106	Actual top of memory (# of pages)
564	Light Pen Horizontal value
565	Light Pen Vertical value

704	Player-missile 0 color
705	Player-missile 1 color
706	Player-missile 2 color
707	Player-missile 3 color
708	Color field 0 color
709	Color field 1 color
710	Color field 2 color
711	Color field 3 color
712	Background color
741,742	Top of available user memory (LSB,MSB)
752	Cursor inhibit (0 = cursor on)
755	Character mode register
756	Character base register (defaults to 224) (224 = upper case, 226 = lower case characters)
763	Last ATASCII character
764	Last keyboard key pressed; int. code;(255 = no key)
765	Fill data for graphic DRAWTO/FILL
766	Display Flag (0 = execute control character)
767	Start/Stop flag for paging (0 = normal listing)
794	Handler address table (3 bytes/handler)
832	I/O control blocks (16 bytes/IOCB)
1664-1791	Spare RAM

BASIC PEEKS (NO POKES!)

186-187	Line # at which STOP occurred
195	Error # for display to user
201	Print tab width (defaults to 10)
212,213	Low and high bytes of value to be returned to BASIC from USR function
251	RAD/DEG flag (0 = radians, 6 = degrees)

3. Most programs can be **RUN** using the computer console, the ATARI 410™ Program Recorder, and any good quality (preferably color) TV set. However, you should consider the various optional ATARI peripherals such as the ATARI 810™ Disk Drive, the ATARI 820™ 40-column or ATARI 825™ 80-column Printers, and the ATARI 830™ Modem. While we're getting this "word from our sponsor", we should also mention that additional plug-in RAM memory is available in 8K or 16K increments (ATARI 800 only). The 400 must be upgraded by an ATARI service facility, and its RAM may be increased up to a total of 16K.

We think you'll find that these accessories and peripherals will save you programming time, and will allow even greater flexibility in the use of your total ATARI system.

4. Try to bear in mind the nature of computers when planning and writing your programs. Computers are unique among machines, for they have no specific function, but instead can be programmed to do literally thousands of different things. Programs that are well-defined and planned allow the computer to maximize its efficiency in performing its many-faceted tasks.

5. Finally, try to remember that a **Human Being** (you or someone else), will have to deal with your programmilng efforts. Most programs allow the computer to communicate with a user (or player, in the case of game programs). If the user must rely on his very fallible human memory, errors will result that will be blamed on your pride and joy! Provide lots of "prompts" that will tell the user or player what to do next. Build in "error traps", too, that will tell the forgetful human what he did wrong. Also, remind yourself periodically that human beings, even programers, have one thing that no computer (yet) possesses: A sense of humor!

And that completes our tour of BASICland! Hope you learned a lot and enjoyed yourself. There's one thing that sets **this** tour apart: You can always come back, and by picking up this book, look up material you missed the first time around or have forgotten. Use the tour book as a reference, and we feel that you will find the answers to almost all of your programming questions. You may want to start your own notebook of computer-oriented material such as the newsletters that ATARI mails out to owners who have returned warranty cards on systems and peripherals. Every scrap of knowledge you can glean from **any** reliable source will make you a better programmer. And you'll be amazed to find that your own thinking processes becomes less cluttered and more straight-forward because you know how a computer would go about solving a particular problem! You'll soon realize that you don't really know how to do something unless you can write a program that will let a computer do it. Your personal computer will literally change your whole pattern of life! Wait and see...

MEANWHILE, JUST ENJOY!

APPENDIX A

A COLLECTION OF PROGRAMS YOU CAN "PUNCH UP" AND RUN

The programs and routines in this appendix were selected to allow the new computerist to try out his ATARI 400 or 800 Personal Computer System, and to give him some idea of its tremendous possibilities. Most of these programs are "bare bones" routines that fairly cry out to be "embellished". For example, the routines that use **SOUND** could also incorporate color graphics and vice versa.

Just key the routine's into your Atari Personal Computer, **RUN** them as often as you like, and if you find some that you may want to **RUN** in the future, record them on disk or cassette.

Whatever else you may do with these routines, **please do enjoy yourself!**

USER PROGRAM #1
PROGRAM TO CHECK HEARING/HEARING AIDS

1. Type this program into your ATARI 400/800:

```
5 GRAPHICS 0
10 FOR S = 1 TO 40
20 PRINT S
30 SOUND 0, S, 10, 10
40 FOR N = 1 TO 1000: NEXT N
50 NEXT S
60 PRINT "TEST COMPLETE - RECORD NUMB
ER FIRST HEARD"
```

2. Now "RUN" with your hearing aid (if any) off. Record the number from the screen that you saw **when you first heard the very high pitched sound.**

3. *Repeat* the test with your hearing aid (if you wear one) on and operating. Again, record the number from the screen when you **first** heard the tone.

4. **Compare** the two numbers. The higher the number, the lower the sound you were first able to hear.

NOTE: A person with "normal" hearing can usually hear the note at numbers 3-6.

5. If there's a **significant** difference between what you hear with and without your hearing aid, this would seem to indicate that it is functioning at least to some extent.

USER PROGRAM #2
ALPHA-NUMERIC SORTING ROUTINE USING LEN FUNCTION

About this program: This program allows you to input (by keyboard) as many numbers or words as you want (entries may be mixed numbers and words, too!), and the computer will then put all the numbers in numerical order and all the words in alphabetical order. The numbers will be printed out first, followed by the words.

Note: If you don't have a printer, simply omit the starred (*) lines in the program, and the monitor screen will then carry the output! The ordered items will be printed out in a vertical column unless more than 20 are entered (which runs you out of room on the TV screen). In this case, they will be printed out across the screen in the normal manner.

PROGRAM LIST	COMMENTS
`10 GR. 0: PRINT "  ALPHANUM ERIC SORT":PRINT`	Clear screen, program title (6 spaces between" and ALPHANUMERIC)
`20 DIM A$(2000), B$(50), C$(50):SETCOLOR 2,12,2: A$=""`	DIM the 3 strings, set color register 2, make A$ a null string
`30 PRINT "HOW MANY ITEMS TO SORT?":INPUT W`	Asks for number of entries
`35 W = W+1`	Increment items by 1
`40 FOR I = 1 TO W`	Start looping W. times
`50 IF I=1 THEN B$=" ": GOTO 80`	Skip to 80 if I is 1
`60 PRINT:PRINT "ENTER AN ITEM PLEASE "`	Asks for an entry
`70 INPUT B$`	Takes entry, assigns to B$
`*75 LPRINT B$`	PRINT B$ on printer
`80 B$(LEN(B$)+1)=" (10  )"`	Initialize to blanks
`90 A$(LEN(A$)+1)=B$(1,10)`	Tack B$ onto end of A$
`100 NEXT I`	Go back to 40 for next I value
`110 FOR I=11 TO 10*W STEP 10`	Start loop with I set from ten times W
`120 FOR L=11 TO 10*W STEP 10`	Next second loop inside first
`130 IF A$(L,L+9)<A$(L-10,L) THEN GOSUB 1000`	If sort required, goto subrout.
`140 NEXT L: NEXT I`	Completes both loops
`150 GR.0`	Clear screen by "changing" modes
`160 IF W>20 THEN 200`	If more than 20 times goto 200
`170 GOSUB 2000`	Jump down to subroutine at 2000
`200 PRINT A$`	Print out the sorted items
`*205 LPRINT: LPRINT: LPRINT`	3 blank lines on printer
`*210 LPRINT A$`	Hard copy print of sorted items
`220 GOSUB 3000`	Jump to subroutine at line 3000
`999 END`	Keeps program separated from subroutine
`1000 C$= A$(L,L+9)`	Assign A$ (10 spaces) to C$
`1010 A$(L,L+9)=A$(L-10,L)`	Move spaces in A$ back 10 spaces
`1020 A$(L-10,L)=C$`	C$ now follows characters moved back
`1030 RETURN`	Jump back to main program
`1999 END`	(See 999)
`2000 POKE 82,11: POKE 83,20`	Set margins so words are output 1 at a time in vertical column
`2010 RETURN`	
`2999 END`	
`3000 POKE 82,3: POKE 83,36`	Return margins to normal setting
`3010 RETURN`	

*Note: If you are not using a printer, then eliminate the starred lines (75, 205, 210). This will give you screen output.

USER PROGRAM #3
CHECKBOOK BALANCER PROGRAM

This is one of the "traditional" programs that every beginning computerist writes. You might want to try your hand at this one first, and if you get "stuck" or if you want to compare your efforts with someone else's, then have a look at this one. Basically, this little program just takes your old balance, asks you to input your checks by number and amount, and then uses an entered "0" to indicate that you are through entering checks. It then does the same for deposits. All entries are put in a three column matrix, where column 1 is the check or deposit number, column 2 is the amount, and column 3 is the computed balance.

If you have someone in your family who laughs at your computer and wonders about its practical value (if any), key this one up and show it to him! Then CSAVE it for use later.

PROGRAM LISTING	COMMENTS
`10 GR.0: ? " CHECKBOOK BALANCER": ?`	9 spaces between 1st "and **CHECKBOOK**"
`20 DIM A(100,3)`	Matrix is 100 rows by 3 columns
`30 ? "Enter 0 after last check or depos it is in the computer":?" R=0`	Note lower case printing.
`40 SETCOLOR 2,11,2`	Make background green
`100 REM INPUT OLD BALANCE`	Remarks help document the program!
`110 ? "OLD BALANCE?": INPUT OB: A(R,3)=OB: ?: R=1`	R is counter variable
`200 REM INPUT CHECKS BY NUMBER & AMOUNT`	More documentation
`210 TRAP 210`	If error in input, return to 210
`220 ?:?"CHECK NUMBER?": INPUT CN`	Ask for number, take input
`230 IF CN=0 THEN 300`	If 0 is entered, goto 300
`240 ? "AMOUNT OF CHECK?": INPUT CA`	Enter a check
`250 A(R,1)=CN: A(R,2)=CA`	Assign check to matrix variables
`260 REM GET RUNNING BALANCE IN COL. 3`	Figure the balance
`270 A(R,3)=A((R-1),3)-A(R,2)`	Subtract check from old balance
`280 R=R+1:T=R:GOTO 210`	Increment counter, do it again.
`300 REM INPUT DEPOSITS`	Remarks to document program
`310 TRAP 310`	Faulty input? Do it again!
`320 ?:?"IF LAST DEPOSIT DONE, TYPE 0, OTHERWISE TYPE 1": INPUT DN`	Enter 0 when done with deposits
`325 IF DN=0 THEN 400`	If all entered, jump to printout
`330 ? "AMOUNT OF DEPOSIT?":INPUT DA:A(R,2)=DA:A(R,1)=0`	Get deposit, assign to matrix
`340 REM GET RUNNING BALANCE IN COL. 3`	more documentation!
`350 A(R,3) = A((R-1),3+A(R,2)`	Add deposit to old balance
`360 R=R+1:T=R:GOTO 310`	Increment counter, do it again

```
400 REM PRINT OUT ENTIRE MATRIX
410 GR.0:?"OLD BALANCE WAS ";OB:?
420 ? "NUMBER","AMOUNT","BALANCE":?
430 FOR R=0 TO T-1
440 ? A(,R1),A(R,2),A(R,3)
450 NEXT R
460 ?:? "NOTE: 0 in column 1 indicates a
deposit.
470 END
```

And still more comments!
Clear screen, begin print out
Headings for 3 column output
So *that's* what T and R do...!
Print out each matrix element

This line is optional...END is required in many BASICs...

Hope your deposits exceed your checks!

USER PROGRAM #4
SUBROUTINE FOR EXTRACTING "EVEN" CUBE ROOTS

Sooner or later, you will run into a program that wants the cube root of a number *only* if it "comes out even", that is, without decimals. This is a little subroutine that will do the job for you. It has been kept as simple as possible, and no attempt has been made to "speed up" the execution by eliminating certain numbers, numbers above a certain value, etc. We leave that up to you. Besides, the slow execution time is kind of fun, and will give you a new appreciation for the many Functions in BASIC that do this kind of thing for you almost instantaneously (by machine-language routines).

PROGRAM LIST

```
10 GR.0


20 PRINT " WHAT IS THE NUMBER"
30 INPUT X
40 GOSUB 9000
8999 END

   REM CUBE ROOT SUBROUTINE
   FOR Y= 1 TO INT(0.3 * X)
   IF Y =INT(0.3 *X) THEN PRINT "NO EV
 CUBE ROOT": END
   IF X=Y*Y*Y THEN POP: GOTO 9040
   NEXT Y
   PRINT "THE CUBE ROOT OF ";X;"IS";

   RETURN
```

COMMENTS, ETC.

Clear the screen. Note this part of the program just "calls" the subroutine that starts at line 9000

Assign the number typed to variable X
Call" the subroutine at line 9000
Prevents calling program from "crashing" into the subroutine
Documents what routine does
Try numbers up to Approx 1/3 X
If .3*X is reached, print out & quit
See Chapter 3 for more on

USER PROGRAM #5
"LIGHT SHOW"

Here's a little graphics routine that demonstrates most of the ATARI Mode 7 color Graphics capabilities, is fun all by itself, or can serve as a "nugget" of a program *you* create. It'll take you about 5 minutes to key in (there are only 7 lines). An interesting programming feature is the use of one FOR/NEXT loop to control the STEP variable of another loop, while the color selected is controlled by the same variable! Hope this one leads you on to all kinds of creative embellishments!

PROGRAM LIST	PROGRAMMER'S COMMENTS
10 FOR ST=1 TO 8:GR. 7	Start loop for step variable (ST)
20 ?:?" *Atari's Special Light Show!*": SETCOLOR 2,0,0	Use reverse video lower case where text is in *italics*. Make background & text window black
30 SETCOLOR 1, 2*ST, 8: COLOR 2	Vary plotted color with ST
40 FOR DR=0 TO 80 STEP ST	ST is the step variable changed in 10
50 PLOT 0,0:DRAWTO 100,DR	Draw the pattern on the screen
60 NEXT DR:FOR N=1 TO 800:NEXT N: NEXT ST	All loops now properly nexted and closed.
70 FOR N=1 TO 2000:NEXT N: GOTO 10	Delay for 2000 count, then start over.

Note use of **lower** case letters and **inverse video** in line 20. You can do this any time in a print statement when the characters involved are between quotation marks!

USER PROGRAM #6
TYPE-A-TUNE PROGRAM

This is a little piece of pure fun, and totally without redeeming social importance! It lets you play music by pressing the keys (top line of the keyboard). The keyboard is scanned, and the letter pressed is compared with the contents of two matrices. If a match is found, the correct note is produced by the SOUND statement in line 130. this is a "bare bones" program that needs your own creative input!

PROGRAM LIST	COMMENTS, ETC.
5 GRAPHICS 0	
10 DIM CHORD(37)	
20 DIM TUNE(12)	DIM the variables CHORD and TUNE.
30 FOR X=1 TO 37:READ A: CHORD(X)=A: NEXT X	Load matrix CHORD from DATA statements
40 FOR X=1 TO 12: READ A: TUNE(X)=A: NEXT X	Load matrix TUNE from DATA statements
50 OPEN #1, 4, 0, "K:"	OPEN the keyboard for input as device #1 (See Ch. 5).
60 IF PEEK(764)=255 THEN 60	Check for keystroke: If none, repeat.
70 A=PEEK(764)	If there has been a keystroke, store its code in A.
80 Z=PEEK(53775)	Store contents of location 53775 in Z
90 X=1	Do we **have** to say more?
100 IF TUNE(X)=A THEN 130	Look through matrix TUNE for note played. If found go to 130 and play it. If not, try next element.
110 X=X+1:IF X=13 THEN 60	
120 GOTO 100	
130 SOUND 0, CHORD(X), 10, 8	The SOUND command!
140 IF PEEK(53775)<>Z THEN 170	If this has changed, go to 170
150 IF PEEK(764)<>A THEN 70	If this has changed, go back to 70
160 GOTO 140	Has anything changed *yet*?
170 POKE 764,255	If Z has changed, return 764 to 255, which is the "no keystroke" code.
180 SOUND 0,0,0,0	Turn off the music machine! Note over.
190 GOTO 60	
180 SOUND 0,0,0,0	Turn off the music machine! Note over.
190 GOTO 60	Loop back to 60 and look for next note.
200 DATA 243,230,217,204,193, 182,173,162,153,144,136,128,121,114,10 8,102,96,91,85,81,76,72,68,64,60	The three DATA statements hold the pitch values for the various notes.
210 DATA 57,53,50,47,45,42, 40,37,35,33,31,29	
220 DATA 31,30,26,24,29,27,51, 53,48,50,54,55	

USER PROGRAM #7
"GRAPHITI"

This is a 1 to 4 player game that allows the players to draw lines and designs on the screen in up to 4 colors (white, red, blue, and black). It uses the joysticks, and provides some excellent examples of the GRAPHICS modes and the game controller commands (STICK, STRIG, etc.) We think you'll enjoy the game itself and will learn from working with the BASIC program!

PROGRAM LIST	COMMENTS, ETC.
10 GRAPHICS 0	Clear the screen
20 ? " VIDEO GRAPHITI"	Print the program title
30 REM ARRAYS X & Y HOLD COORDINATES	Documenting program with remarks
40 REM FOR UP TO 4 PLAYER POSITIONS	
50 REM COLR ARRAY HOLDS COLORS	
60 DIM A$(1),X(3),Y(3),COLR(3)	DIMension the 3 arrays and 1 string
70 ? "Use Joysticks to Draw Pictures"	
80 ? "Press Trigger Buttons to Change Colors"	Instructions to players
90 ? "INITIAL COLORS:"	
100 ? "Joystick 2 is White"	Print color info to players
110 ? "Joystick 3 is Blue"	
120 ? "Joystick 4 is Background Color"	
130 ? "INITIAL COLORS:"	
140 ? "BLACK LOCATION INDICATED BY BRI EF FLASH OF RED"	
150 ? "IN GRAPHICS 8, JOYSTICKS 1 & 3 ARE WHITE AND 4 IS BLUE"	
160 ? "How many players (1-4)?"	
170 INPUT A$:IF LEN(A$)=0 THEN A$="1"	
180 JOYMAX=VAL(A$)-1	
190 IF JOYMAX<0 OR JOYMAX>=4 THEN 138	JOYMAX & JOYIN are variable names
200 ? "GRAPHICS 3 (40X24), 5(80X40),"	
210 ? "7 (160X96), OR 8 (320X192)?";	
220 INPUT A$:IF LEN(A$)=0 THEN A$="3"	
230 A=VAL(A$)	
240 IF A=3 THEN XMAX=40:YMAX=24:GOTO 290	Establishing "wraparound" values
250 IF A=5 THEN XMAX=80:YMAX=48:GOTO 290	
260 IF A=7 THEN XMAX=160:YMAX=96:GOTO 290	
270 IF A=8 THEN XMAX=320:YMAX=192:GOTO 290	

```
280 GOTO 200:REM A NOT VALID
290 GRAPHICS A+16
300 FOR I=0 TO JOYMAX:X(I)=XMAX/2+1:Y(
I)=YMAX/2+1:NEXT I:REM START NEAR CENTER
OF SCREEN
310 IF A<>8 THEN 350
320 FOR I=0 TO 2:COLR(I)=I+L:NEXT I
330 SETCOLOR 1,9,14:REM LIGHT BLUE
340 GOTO 380
350 FOR I=0 TO 2:COLR(I)=I+L:NEXT I
360 SETCOLOR 0,4,6:REM RED
370 SETCOLOR 1,0,14:REM WHITE
380 COLR(3)=0
390 FOR J=0 TO 3
400 FOR I=0 TO JOYMAX:REM CHECK JOYSTI
CKS
410 REM CHECK TRIGGER
420 IF STRIG(I) THEN 470
430 IF A<>8 THEN 460
440 COLR(I)=COLR(I)+1:IF COLR(I)=2 THE
N COLR(I)=0:REM TWO-COLOR MODE
450 GOTO 470
460 COLR(I)=COLR(I)+1:IF COLR(I)=4 THE
N COLOR (I)=0:REM FOUR-COLOR MODE
470 IF J>0 THEN COLOR COLR(I):GOTO 500
480 IF COLR(I)=0 THEN COLOR 1:GOTO 500
490 COLOR 0:REM BLINK CURRENT SQUARE O
N AND OFF
500 PLOT X(I),Y(I)
510 JOYIN=STICK(I):REM READ JOYSTICK34
0 IF JOYIN=15 THEN 530:REM NO MOVEMENT
520 JOYIN=STICK(I):REM READ JOYSTICK34
0 IF JOYIN=15 THEN 530:REM NO MOVEMENT
530 COLOR COLR(I):REM MAKE SURE COLOR
IS ON
540 PLOT X(I),Y(I)
550 IF JOYIN>=8 THEN 600
560 X(I)=X(I)$1:REM MOVE RIGHT
 '0 REM IF OUT OF RANGE THEN WRAPAROUN
D
580 IF X(I)=XMAX THEN X(I)=0
590 GOTO 630
600 IF JOYIN>=12 THEN 630
```

Choose Graphics Mode A without Text Window

If A is not GR.8, go to line 350

Setting color registers

Color control statements

Reading joystick positions and plotting the Graphics points on the screen

Control of "wraparound" game feature

```
610 X(I)=X(I)-1:REM MOVE LEFT
620 IF X(1)<0 THEN X(I)=XMAX-1
630 IF JOYIN<>5 AND JOYIN<>9 AND JOYIN
<>13 THEN 660
640 Y(I)=Y(I)+1:IF Y(I)=YMAX THEN Y(I)=
0:REM MOVE DOWN
650 GOTO 680
660 IF JOYIN<>6 AND JOYIN<>10 AND JOYI
N<>14 THEN 680
670 Y(I)=Y(I)-1:IF Y(1)<0 THEN Y(I)=YM
AX-1:REM MOVE UP
680 PLOT X(I),Y(I)
690 NEXT I
700 NEXT J
710 GOTO 390
```

APPENDIX B

ALPHABETICAL DIRECTORY OF BASIC RESERVED WORDS

RESERVED WORD:	ABBREVIATION:	SEE CHAPTER:	BRIEF SUMMARY OF BASIC STATEMENT
ABS		6	Function returns absolute value (unsigned) of the variable or expression.
ADR		6	Function returns memory address of a string or a numeric array.
AND		4	Logical operator: Statement is true only if both substatements joined by **AND** are true.
ASC		7	String Function returns the numeric value of a single string character.
ATN		6	Function returns the arctangent of a number or expression in radians or deg
BYE	**B.**		Exit from BASIC & return to the resident operating system or console processor
CLOAD	**CLOA.**	5	Loads data from Prog[illegible] Re[illegible]order into RAM. Includes the **OPEN** statement
CHR$		7	String function returns a [illegible]gie string byte equivalent to a numeric valu[illegible] betw[illegible]er 0 and 255 in ATASC11 code.
CLOSE	**CL.**	5	I/O statement [illegible]sed t[illegible] close a file at the conclusion of I/O operation
CLR		8	The opposite [illegible] **DIM** [illegible]nD[illegible]M[illegible]ensio[illegible]a[illegible]str[illegible]gs; matrices.
COLOR	**C.**	9	Chooses co[illegible]or re[illegible]ist[illegible] [illegible][illegible] [illegible]or graphics work
CONT	**CON.**	2	Continue[illegible] causes [illegible]prog[illegible] [illegible] execution [illegible]n the next line follo[illegible] [illegible] [illegible] ev[illegible] [illegible]cuntering a **STOP** in the prog[illegible]m
[illegible]S		6	Function [illegible]et[illegible]s the cosi[illegible] [illegible]f [illegible] expression (degrees or [illegible]adians)
[illegible]		5	Outputs [illegible] [illegible]RAM [illegible] [illegible] R[illegible]e[illegible]or tape st[illegible] [illegible] [illegible]nt
[illegible]	**E.**	5	I/O com[illegible] [illegible] [illegible] [illegible]n [illegible] [illegible]kenize[illegible] c[illegible]

RESERVED WORD	ABBREVIATION	SEE CHAPTER	BRIEF SUMMARY OF BASIC STATEMENT
DATA	**D.**		Part of **READ...DATA** combination. Used to identify the succeeding items (which must be separated by commas) as individual data items.
DEG	**DE.**	6	Statement **DEG** tells computer to perform trigonometric functions in degrees instead of radians. (Default is radians)
DIM	**DI.**	8	Reserves the specified amount of memory for matrix, array, or string. All strilng variables, arrays, matrices must be **DIMed**.
DOS	**DO.**	(see DOS Manual)	Reserved word for disk operations. Causes the menu to be displayed.
DRAWTO	**DR.**	9	Draws a straight line between a **PLOT**ted point and specified point.
END		2	Stops program execution. Program may be restarted using **CONT** (Note: **END** may be used more than once in a program)
EXP		6	Function returns e (2.7182818) raised to the specified power.
FOR	**F.**	3	Used with **NEXT** to establish **FOR...NEXT** loops. Introduces the range that the loop variable will operate in during the execution of loop.
FRE		6	Function returns the amount of user memory remaining (in bytes.)
GET	**GE.**	5,9	Used mostly with disk operations to input a single byte of data.
GOSUB	**GOS.**	3	Branch to a subroutine beginning at the specified line number.
GOTO	**G.**	3	Unconditional branch to a specified line number.
GRAPHICS	**GR.**	9	Specifies which of the eight graphics modes is to be used. **GR.0** may be used to clear screen.
IF		3	Returns a 1 only if the statement is true, otherwise returns 0. May be used to cause conditional branching or to execute another statement on the same line (only IF the first statement is true).

INPUT	**I.**	5	Causes computer to ask for input from keyboard. Execution continues only when RETURN key is pressed after inputting data. (Pressing RETURN without keyboard-entered data is an error.)
INT		6	Function returns the next lowest whole integer below the specified value. Rounding is *always* downward, even when number is (−).
LEN		7	String function returns the length of the specified string in bytes or characters (1 byte contains 1 character).
LET	**LE.**	4	Assigns a value to a specific variable name. **LET** is optional in ATARI BASIC, and may be simply omitted!
LIST	**L.**	2	Display or otherwise output the program **LIST.**
LOAD	**LO.**	5	Input from disk, etc. into the computer. Included **OPEN.**
LOCATE	**LOC.**	9	Graphics: Stores in a specified variable, the color register # that controls a specified graphics point.
LOG		6	Function returns the natural logarithm of a number
LPRINT	**LP.**	5	"Shortcut" command to lineprinter to print the specified message. Includes the **OPEN** statement.
NEW		2	Erases all contents of user RAM. **Be careful with NEW!!!**
NEXT	**N.**	3	Causes a **FOR...NEXT** loop to terminate or continue depending on the particular variables or expressions. All loops are executed at least once.
NOT		4	A "1" is returned only if the statement is NOT true. If it *is* true, a "0" is returned.
NOTE	**NO.**		See **DOS/FMS Manual**...used only in disk operations.
ON		3	Used with **GOTO** or **GOSUB** for branching purposes. Multiple branches to different line numbers are possible depending on the value of the **ON** variable or expression.
OPEN	**O.**	5	Opens the specified file for input or output operations.
ON		4	Logical operator used between two statements. If either one is true, a "1" is evaluated. A "0" results only if both are false.
PADDLE		10	Function returns position of the paddle game controller.
PEEK		11	Function returns decimal form of contents of specified memory location (RAM or ROM).
PLOT	**PL.**	9	Causes a single point to be plotted at the X,Y location specified.

RESERVED WORD	ABBREVIATION	SEE CHAPTER	BRIEF SUMMARY OF BASIC STATEMENT
POINT	**P.**	(See DOS/FMS Manual)	Used with Disk operations only.
POKE	**POK.**	11	Insert the specified byte into the specified memory location. May be used only with RAM. Don't try to **POKE** ROM! You'll get an ERROR!
POP			Removes the loop variable from the **FOR/GOSUB** stack. Used when departure from the loop is made in other than normal manner.
POSITION	**POS.**	9	Sets the cursor to the specified screen position.
PRINT	?	5	I/O command causes output from the computer to the specified output device. Default is to the TV screen.
PTRIG		10	Function returns status of the trigger button on game controllers.
PUT	**PU.**	5, 9	Causes output of a single byte of data from the computer to the specified device.
RAD		6	Specifies that information is in radians rather than degrees when using the trigonometric functions. Default is to **RAD**. (See **DEG**).
READ	**REA.**		Read the next item in the **DATA** list and assign to specified variable.
REM	**R.**	2	Remarks: This statement does nothing, but comments may be printed within the program **LIST** for future reference by the programmer. Statements on a line that starts with **REM** are **not** executed.
RESTORE	**RES.**		Allows **DATA** to be **READ** more than once.
RETURN	**RET.**	3	**RETURN** from subroutine to the line immediately following the one on which **GOSUB** appeared.
RND		6	Function returns a random number between 0 and 1, but never 0 or 1.
RUN	**RU.**	2	Execute the program. Sets normal variables to 0, un**DIM**s arrays, strings; string variables set to ATASCII 0 ( ) when **DIM**ed.
SAVE	**S.**	5	"Shortcut" I/O statement causes data or program to be recorded on disk under filespec provided with **SAVE** Includes **OPEN.**

SETCOLOR	**SE.**	9	Store hue and luminance color data in a particular color register.
SGN		6	Function returns +1 if value is positive, 0 if zero, −1 if negative
SIN		6	Function returns trigonometric sine of given value (**DEG** or **RAD**)
SOUND	**SO.**	10	Controls register, sound pitch, distortion, and volume of a tone or note.
SQR		6	Function returns the square root of the specified value.
STEP		3	Used with **FOR...NEXT**. Determines quantity to be skipped between each pair of loop variable values.
STICK		10	Function returns position of stick game controller.
STRIG		10	Function returns 1 if stick trigger button not pressed, 0 if pressed.
STOP	**STO.**	2	Causes execution to stop but does not reset arrays, other variables
STR$		7	Function returns a character string equal to numeric value given. For example: STR$ (65) returns 65 as a string.
THEN		3	Used with **IF**: If statement is true, the **THEN** statement is executed. If the statement is false, control passes to next line.
TO		3	Used with **FOR** as in "FOR X = 1 TO 10". Separates the loop range expressions.
TRAP	**T.**	3	Takes control of program in case of an **INPUT** error and directs execution to a specified line number.
USR		11, App. I	Function returns results of a machine-language subroutine to the specified variable.
VAL		7	Function returns the equivalent numeric value of a string.
XIO	**X.**	5	General I/O statement used with disk operations (see **DOS/FMS Manual**) and in graphics work (**FILL**).

Note The period is *mandatory* after all abbreviated keywords.

APPENDIX C

ERROR MESSAGE NUMBER CODES

2 **Memory insufficient** to store the statement or the new variable name or to DIM a new string variable.

3 **Value Error:** A value expected to be a positive integer is negative, a value expected to be less than 256 is not, or a value that was expected to be within a specific range is not.

4 **Too Many Variables:** A maximum of 128 different variable names is allowed.

5 **String Length Error:** Attempted to store beyond the DIMensioned string length.

6 **Out of Data Error:** READ statement requires more data items than supplied by DATA statement(s).

7 **Number greater than 32768:** Value is not a positive integer or is greater than 32768.

8 **Input Statement Error:** Attempted to INPUT a non-numeric value into a numeric variable.

9 **Array or String DIM Error:** DIM size is greater than 32767 or an array/matrix reference is out of the range of the DIMensioned size, or the array/matrix or string has been already DIMensioned.

10 **Argument Stack Overflow:** There are too many GOSUBs or too large an expression.

11 **Floating Point Overflow Error:** Attempted to divide by zero or refer to a number larger than $1x10^{98}$ or smaller than $1x10^{-99}$.

12 **Line Not Found:** A GOSUB, GOTO, or THEN was referenced to a non-existent line number.

13 **No Matching FOR Statement:** A NEXT was encountered without a previous FOR, or nested FOR...NEXT statements do not match properly. (Error is reported at the NEXT statement, not at FOR).

14 **Line Too Long Error:** The statement is too complex or too long for BASIC to handle.

15 **GOSUB or FOR Line Deleted:** A NEXT or RETURN statement was encountered and the corresponding FOR or GOSUB has been deleted since the last RUN.

16 **RETURN Error:** A RETURN was encountered without a matching GOSUB.

17 **Garbage Error:** Execution of "garbage" (bad RAM bits) was attempted. This error code **may** indicate a hardware problem, but may also be the result of faulty use of POKE. Try typing NEW or powering down, then re-enter the program without *any* POKE commands.

18 **Invalid String Character:** String does not start with a valid character, or string in VAL statement is not a numeric string.

NOTE: **The following are INPUT/OUTPUT errors that result during the use of disk drives, printers, or other accessory devices. Further information is provided with the auxiliary hardware.**

19 **LOAD Program Too Long:** Insufficient memory remains to complete LOAD.

20 Device Number Larger Than 7.

21 LOAD File Error: Attempted to LOAD a non-LOAD file.

128 BREAK Abort: User hit BREAK key during I/O operation.

129 IOCB Already Open.

130 Nonexistent Device specified.

131 IOCB Write Only. READ command to a write-only device (printer).

132 Invalid Command: The command is invalid for this device.

133 Device of File not Open: No OPEN specified for the device.

134 Bad IOCB Number: Illegal device number.

135 IOCB Read Only Error: WRITE command to a read-only device.

136 EOF: End of File read has been reached. (NOTE: This message may occur when using cassette files.)

137 Truncated Record: Attempt to read a record longer than 256 characters.

138 Device Timeout... Device doesn't respond

139 Device NAK: Garbage at serial port or bad disk drive

140 Serial bus input framing error

141 Curser out of range for particular mode

142 Serial bus data fram overrun

143 Serial bus data frame checksum error

144 Device done error (invalid "done" byte)

145 Read after write compare error (disk handler)

146 Function not implemented in handler

147 Insufficient RAM for operating selected Graphics Mode

160 Drive # error

161 Too many OPEN files (no sector buffer available)

162 Disk full (no free sectors)

163 Unrecoverable system data I/O error

164 File # mismatch

165 File name error

166 POINT data length error

167 File locked

168 Command invalid (special operation code)

169 Directory full (64 files)

170 File not found

171 POINT invalid

Words to Live by:

"*Computers* do not make errors! Computers only *report* errors made by human beings. Humans are notoriously imperfect..."

PROGRAMMING IN MACHINE LANGUAGE

What Is Machine Language?

If BASIC is a "high level" language, then machine language is the lowest! Machine language is written entirely in binary code, meaning that only the digits 0 and 1 may be used. We can think of a 0 as being something "false" (or something true!) Same with 1! Or we can represent numbers in binary by saying that binary 0 is the same as decimal 0, 001 is 1, 010 is 2, 011 is 3, 100 is 4, 101 is 5, 110 is 6, and so forth. The limit of our count is the number of binary 1's and 0's (called "bits") we have available. Most computers (like yours), use 8 bit bytes, or words. This means that any number up to 11111111 (eight 1's) in binary (or decimal 255) may be expressed in a single byte.

We can also say that 0 indicates a circuit or part that is switched "off" while 1 represents a circuit that is "on". We can (and do) represent letters and other symbols with code numbers (See the ATASCII Code in Appendix E). Indeed, when broken down to the lowest common denominator, everything your computer does, it does in binary digits or bits, each of which may be either a 1 or a 0!

Hexadecimal:

Hexadecimal, or **base 16** numbers are also used a lot with computers because it is relatively easy for us poor humans to think in hex (compared with all those dumb 1's and 0's!) and more important, because hex converts easily and directly to binary! Hex is just like decimal (if you're missing four toes or have six extra fingers!) It goes like this: 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Obviously, the A represents 10, B is 11... F is 15. 16 in hex is represented just like in decimal: By starting over again at 1 and adding a zero to get 10. But don't worry about learning to convert hex to decimal. You won't need to because we've included a handy table in Appendix D, along with a simple conversion program so your computer can do the work! (Humans should not stoop to this kind of mathematical scut-work!) As for converting to binary and vice-versa, just remember that groups of four bits can be converted easily in the head. 1111 is F in hex. 0011 is 3, and so forth. Here's a simple table you can use:

Binary	Hexadecimal	Decimal
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

The simplicity of converting the hex system to and from binary should now be obvious, and the fact that the highest number representable by four bits (half a byte) is also representable as the highest number possible with a single hex digit (why don't we call it a "hit" if a binary digit's a "bit"?) is very useful. Obviously, hexadecimal FF (binary 11111111) is the largest number we can represent using only one byte, (8 bits). The same number in decimal form is 255.

Theoretically, then, we could write a program using groups of 1's and 0's. Doesn't sound like much fun, though. Even hex codes aren't! But at least it's **possible** to "hand assemble" a program using hex. You can do this if you're willing to study the 6502 microprocessor chip and its instruction set. The result of this kind of program (**way** beyond the scope of a **BASIC Reference Manual** like this one) is a whole bunch of hex numbers which now somehow must be fed to your computer. Two problems immediately rear their ugly heads. Where in memory will they go? (We **must** know so we can feed the address to the **USR** function when we **RUN** the program.) And how can the computer use **hex codes** when the operating system converts everything to **floating point, binary coded decimal** notation? Sounds like we can learn to hand assemble a program using the 6502 instruction set, which will give us hexadecimal machine code, but we can't get it into our computer to **RUN** it!

Of course we can convert each hex number to decimal...Hey, wait a minute! Isn't that grist for our computerized mill? You bet a bunch of bits it is! Then what if we entered the converted numbers into a **DATA** statement so we can record the program on tape...we could then **READ** the converted machine codes into an array...but how will we know the addresses of the array elements in actual, physical memory? Why not **POKE** each value into the place in memory where we want it and can find it? Let's see, if we put the **POKE**command and the **READ** command into a **FOR/NEXT** loop...We're even starting to *talk* like real programmers!

We have all kinds of confidence in your programming abilities. But just to save you time, we've developed a simple BASIC program that will do all these things! First it'll let you enter your machine codes in hexadecimal form. It will convert each hex number to decimal, and store the decimal in an array. Finally, the converted codes are printed out in a line that looks for all the world like a **DATA** statement! It starts with the number 1500 just like a line number, then says **DATA** followed by the hex codes you entered, converted to decimals and separated by commas. The final entry on this line is the number 999. The 999 is our "end of file flag", and this just tells the computer that the number before the 999 was the last code in your program. But it isn't really a program line...not quite yet...because **you** the programmer haven't **personally** entered it! Do it by placing the cursor anywhere on the printed **DATA** line, (See Appendix F), and while the cursor is on the line with the DATA, hit RETURN. Now tell it to **LIST 1500**. Son of a gun! You've computed output to the screen, **and caused that output** to become **part of the program that produced the output!** If we had any ham we could have ham and eggs if we had any eggs! Some computer you bought...Next thing you know it'll be programming itself. Then we're **all** in trouble, 'cause it **may** just be smarter than we are!

Take a good look at the program we call Hex Code Loader Input. Then look at the expanded version called "NOTHING" which is really a fully developed example of how this Hex Code

"Loader" program works. This example, when you tell it to GOTO 1000 in direct mode, prints the words "NOTHING IS MOVING" and the machine program changes the colors so fast you'll be amazed. You'll be even more amazed when you see that line 1160 contains a time delay loop! Without it, the colors change so fast that they blend together (to the eye) and can hardly be noticed!

The second example program first does a BASIC graphics design, then shifts into high gear and changes colors using machine code! We think you'll be impressed by the speed, so try it!

We know you're confused, and we realize we can't teach you to program in machine code in a few pages. That's why we recommend that you purchase a copy of the 6502 machine instruction set and the ATARI Assembler-Editor Cartridge[1] (with the book that explains how to write machine/assembly language programs). Meanwhile, do RUN the two examples that follow! We hope they lead you into further study into the incredible speed of the machine's world!

How To Use The Hexcode Loader Program

1. Type NEW and hit RETURN. Then enter the program on the next page, called the "Input Program". When the program is in RAM, type RUN and RETURN.

2. As a test of your program, enter the hexadecimal numbers A, 0, 1, 9, and FF. RETURN after each number. Finally, type "DONE" and RETURN.

3. You should see a printout on the screen that looks like this:

```
1500 DATA 10,0,1,9,255,999
STOPPED AT LINE 230
```

   AT THIS POINT, DO NOT GIVE ANY OTHER COMMANDS. FOLLOW THE INSTRUCTIONS GIVEN BELOW OR YOU MAY LOSE YOUR PROGRAM!!!

4. CSAVE the program on cassette or SAVE it on disk. If you use cassette, remember to write down where on the tape (by the tape counter) the program is! In the future, we will refer to this master program as the Input Program.

5. Now add the BASIC language part of your program, including the USR function that calls the machine language subroutine, beginning at line 1080. Remember the DATA statement will occupy line 1500.

6. Count the total number of hex codes to be entered. Enter this number on line 1000, replacing any number that may already be there. For example, if you have 15 hex code instructions, line 1000 should look like this:

```
1000 CLR: BYTES=15
```

7. Now **CSAVE** or **SAVE** the resident program again. Write down where it is on the tape, if cassette is used. Note that at this point, there is no line 1500! If one is found, eliminate it **before CSAVE** by typing 1500 and then hitting RETURN .

8. You are now ready to enter your machine language subroutine. Tell the computer to **RUN**, and enter the hex codes, one at a time, each one followed by RETURN .

   When the last hex code has been entered and RETURN has been pressed, type the word **DONE** and hit RETURN again. You should now get your print out of line 1500.

9. **Until you instruct it to,** your computer will not accept line 1500 as an actual program line! Once you have read it carefully and are convinced that the hex codes are properly converted to decimal numbers, and that the format is correct for a **DATA** statement, you can absorb the line into your program by moving the cursor (use CTRL with the arrows as described in Appendix F) onto the line occupied by the printout. If there is more than one line, **any line** occupied by a part of the **DATA** printout will do. With the cursor in place, release the control key and RETURN once. Now tell the computer to **LIST 1500** and RETURN again. How about that for computing power? The thing almost wrote it's own program, didn't it?

10. Now **CSAVE** or **SAVE** the completed program. This is **very important**, because if the routin gets into a loop it cannot get out of, you may find yourself with a "locked up" system which you will have to power down to "unlock". Powering down, of course, wipes out your program in RAM. This way you can always reload it again without all that typing...

11. After the program has been recorded on disk or tape, you can execute it and see what happens. To do this, tell it "**GOTO 1000**" in direct mode. Sure is faster than BASIC, isn't it?

Have fun!!!

BASIC Hex-Code Loader Input Program

```
10 GR. 0: ?" HEXACODE LOADER PROGRAM":?
20 REM ALLOWS INPUT OF 6502 HEX CODES.
STORES DECIMAL EQUIVALENTS IN ARRAY A, OUTPU
TS IN PRINTED 'DATA STATEMENT' AT LINE 1500
30 REM USER THEN PLACES CURSOR ON PRIN
TED OUTLINE LINE, STRICKES "RE-
TURN", AND ENTERS REST OF BASIC PROGRAM INC.
USR STATEMENT
40 DIM A(50), HEX$(5)
50 REM INPUT, CONVERSION, STORAGE OF DATA
60 N=0: ? "ENTER ONE HEX CODE. IF LAST ONE IS IN,
ENTER DONE"
```

6 spaces before word HEXCODE

programmer's documentation

Learn to do this!

```
70 INPUT HEX$
80 IF HEX$="DONE" THEN N=999: GOTO 130
90 FOR I=1 TO LEN(HEX$)
100 IF HEX$(I,I) <="9" THEN N=N*16+VAL
(HEX$(I,I)): GOTO 120
110 N=N*16+ASC(HEX$(I,I))-ASC("A")+10
120 NEXT I
130 PRINT N: C=C$1
140 A(C)=N
150 IF N=999 THEN GOTO 200
160 GOTO 60
190 REM PRINT OUT 'DATA LINE 1500"
200 GR.0: ? "1500 DATA";
210 C = 0
220 C=C+1
230 IF A(C)=999 THEN PRINT "999":STOP
240 ? A(C);",";
250 A(C)=0
260 GOTO 220
300 STOP
```

If hexcode is 9 or less, leave it alone.
Otherwise convert to decimal

Print out converted number

End of file flag means job done

Format data line

Stop at end of file flag

NOTE: PROGRAM WILL STOP AT THIS POINT WHILE YOU INCORPORATE THE DATA LINE AT LINE 1500 BY PUTTING CURSOR ON THE PRINTOUT AND HITTING RETURN

```
990 REM EXECUTION MODULE
1000 CLR: BYTES = 00


1010 TRAP 1550: DIM E$(1), E(BYTES)
1020 REM PUT MACHINE CODE IN ARRAY E
1030 FOR I=1 TO BYTES
1040 READ A: IF A>255 THEN GOTO 1060
1050 POKE ADR(E$)+I, A

1060 NEXT I
1070 REM BASIC PART OF USER'S PROGRAM
FOLLOWS (LINES 1080 UP TO 1490)

1550 PRINT "PUT CORRECT NO. OF HEX BYTES IN LINE
1000"
```

(Be sure this number agrees with the no. of hex codes you will enter!)

(Store date in known memory location)

(Put your BASIC program in from 1080 to 1490. Don't forget the USR function)

Note: To execute a completed program, command in direct mode, **GOTO 1000**

Note: Two Sample Machine Code programs on follow.

Sample Machine Code Program #1

Once the BASIC hexcode loader input program is in RAM, you need only do two more things to run the two sample machine programs that follow:.

1. Add the following lines to your BASIC hex code loader input program:

```
1080 GR.7+16
1090 SETCOLOR 0,9,4
1100 SETCOLOR 1,9,8
1110 SETCOLOR 2,9,4
1120 CR=1
1130 FOR X=0 TO 159
1140 COLOR INT(CR)
1150 PLOT 80,0
1160 DRAWTO X, 95
1170 CR=CR+0.125
1180 IF CR=4 THEN CR=1
1190 NEXT X
1200 X=USR(ADR(E$)+1
1210 FOR I=1 TO 15: NEXT I
1220 GOTO 1200
```

HERE ARE THE HEX CODES TO "FEED" THE MACHINE

68	A2	0	AC	C4	2	BD	C5	2	9D	C4
2	E8	E0	2	90	F5	8C	C6	2	60	

(Line 1000 should read **1000 CLR & BYTES = 21**)

2. Feed the machine the actual hex codes. First start the program with the usual **RUN** command. It will ask you for a hex code. After each entry, hit **RETURN**, and the machine will ask you for another one. When all have been entered, type **DONE** and hit **RETURN**. A **DATA** line will be printed out. Now place the cursor a few spaces after the last **DATA** entry number, which will be 999. Press **RETURN**. Now **LIST 1500** and you will see the data that has now been placed in your program. **LIST 1000**. Now note the number following the words **BYTES = _____**. Count the number of entries you made of hex codes, and change the number to match the number of entries. (Do not include the 999 in your count.)

Assembler Subroutin Rotate Colors...

ADDRESS	OBJECT CODE	LINE NO.	LABEL	MNEMONIC	DATA	COMMENTS, NOTES, ETC.
		0100				ROUTINE TO ROTATE COLOR DATA
		0110				FROM ONE REGISTER TO ANOTHER
		0120				3 COLORS ARE ROTATED
		0130	;			
		0140				OPERATING SYSTEM ADDRESSES
02C4		0150				COLOR 0 = $02C$
02C5		0160				COLOR 1 = $02C5
02C6		0170				COLOR 2 = $02C6
		0180	;			
		0190		* =	$6000	MACHINE PROGRAM STARTING ADDRESS
6000	68	0200		PLA		POPSTACK (SEE CHAPTER 11)
6001	A200	0210		LDX	#0	ZERO THE X REGISTER
6003	ACC402	0220		LDY	COLOR0	SAVE COLOR 0
6006	BDC502	0230	LOOP	LDA	COLOR1,X	
6009	9DC402	0240		STA	COLOR0,X	
600C	E8	0250		INX		INCREMENT THE X REGISTER (ADD ONE)
600D	E002	0260		CPX	#2	COMPARE CONTENTS OF X REGISTER WITH 2
600F	90F5	0270		BCC	LOOP	LOOP IF X REGISTER CONTENTS ARE LESS THAN 2
6011	8CC602	0280		STY	COLOR2	SAVE COLOR 0 IN COLOR 2
6014	60	0290		RTS		RETURN FROM MACHINE LEVEL SUBROUTINE

← Assembler Prints This

This Portion is Source Information Programmer Enters Using ATARI Assembler Cartridge →

# Indicates data (source) $ Indicates a Hexadecimal number

Sample Machine Code Program #2

```
1080 GR. 1+16
1090 FOR I=1 TO 6
1100 PRINT #6;"nothing is moving!"          (Use CAPS/LOWR for this)
1110 PRINT #6; "NOTHING IS MOVING!"        (Use SHIFT CAPS/LOWR)
1120 PRINT #6;"nothing is moving!"
1130 PRINT #6;"NOTHING IS MOVING!"
1140 NEXT I
1150 Q=USR(ADR(E$)+1)
1160 FOR I=1 TO 25:NEXT I: GOTO 1150
1170
```

Check that Line 1000 reads like this: **1000 CLR: BYTES = 21**. If it doesn't, fix it

Now enter the hex codes as shown in the following list:

68	A2	0	AC	C4	2	BD	C5	2	9D	C4	2	E8	E0
3	90	F5	8C	C7	2	60							

When done, enter **DONE** and hit RETURN. Now place cursor after last entry (999) on printout **DATA** line and hit RETURN.

Now you can **RUN** your program by typing **GOTO 1000** and hitting RETURN.

Assembler Subroutine to Rotate Colors...

ADDRESS	OBJECT CODE	LINE NO.	LABEL	MNEMONIC	DATA	COMMENTS, NOTES, ETC.
		0100				ROUTINE TO ROTATE COLOR DATA
		0110				FROM ONE REGISTER TO ANOTHER
		0120				4 COLORS ARE ROTATED
		0130	;			
		0140				OPERATING SYSTEM ADDRESSES
02C4		0150				COLOR 0 = $02C$
02C5		0160				COLOR 1 = $02C5
02C6		0170				COLOR 2 = $02C6
02C7		0175				COLOR 3 = $02C7
		0180	;			
		0190		*=	$6000	MACHINE PROGRAM STARTING ADDRESS
6000	68	0200		PLA		POPSTACK (SEE CHAPTER 11)
6001	A200	0210		LDX	#0	ZERO THE X REGISTER
6003	ACC402	0220		LDY	COLOR0	SAVE COLOR 0
6006	BDC502	0230	LOOP	LDA	COLOR1,X	
6009	9DC402	0240		STA	COLOR0,X	
600C	E8	0250		INX		INCREMENT THE X REGISTER (ADD ONE)
600D	E002	0260		CPX	#3	COMPARE CONTENTS OF X REGISTER WITH 2
600F	90F5	0270		BCC	LOOP	LOOP IF X REGISTER CONTENTS ARE LESS THAN 2
6011	8CC602	0280		STY	COLOR3	SAVE COLOR 0 IN COLOR 3
6014	60	0290		RTS		RETURN FROM MACHINE LEVEL SUBROUTINE

← Assembler Prints This →

← This Portion is Source Information Programmer Enters Using ATARI Assembler Cartridge →

# Indicates data (source) $ Indicates a Hexadecimal number

APPENDIX J

PEEKs, POKEs, Pulses and Programs

Note: Many of these locations are of primary interest to *expert* programmers and are included here as a convenience.

Here are a few more hints and bits and pieces of information that may prove useful to your programming efforts.

TABLE OF POTENTIALLY USEFUL MEMORY LOCATIONS

DECIMAL LOCATION	COMMENTS AND DESCRIPTION
14, 15	Highest location used by BASIC (LSB, MSB)
18,19,20	TV frame counter (1/60 sec.) (LSB, NSB, MSB) see above
65	Noisy I/O Flag (O = quiet)
77	Attract Mode Flag (128 = Attract Mode)
82,83	Left, Right Margin (Defaults 2, 39)
84	Current row on which cursor rests (0-23)
85, 86	Current cursor column (0-39)
90	Previous cursor row (0-23)
91, 92	Previous cursor vert. column (0-39)
93	Data under cursor
96	Cursor row to which DRAWTO will go
97,98	Cursor column to which DRAWTO goes
106	Actual top of memory (# of pages)
564	Light Pen Horizontal value
565	Light Pen Vertical value
704	Player-missile 0 color
705	Player-missile 1 color
706	Player-missile 2 color
707	Player-missile 3 color
708	Color field 0 color
709	Color field 1 color
710	Color field 2 color
711	Color field 3 color
712	Background color

741,742	Top of available user memory (LSB,MSB)
752	Cursor inhibit (0 = cursor on)
755	Character mode register
756	Character base register (defaults to 224) (224 = upper case, 226 = lower case characters)
763	Last ATASCII character
764	Last keyboard key pressed; int. code;(255 = no key)
765	Fill data for graphic DRAWTO/FILL
766	Display Flag (0 = execute control character)
767	Start/Stop flag for paging (0 = normal listing)
794	Handler address table (3 bytes/handler)
832	I/O control blocks (16 bytes/IOCB)
1664-1791	Spare RAM
54018	Program Recorder Control

BASIC PEEKS (NO POKES!)

186-187	Line # at which STOP occurred
195	Error # for display to user
201	Print tab width (defaults to 10)
212,213	Low and high bytes of value to be returned to BASIC from USR function
251	RAD/DEG flag (0 = radians, 6 = degrees)