

PROGRAMMING  
YOUR  
ATARI®  
COMPUTER

BY MARK THOMPSON





No. 1453  
\$16.95

# PROGRAMMING YOUR ATARI® COMPUTER

BY MARK THOMPSON



**TAB BOOKS Inc.**

BLUE RIDGE SUMMIT, PA. 17214

FIRST EDITION

FIRST PRINTING

Copyright © 1983 by TAB BOOKS Inc.

Printed in the United States of America

Reproduction or publication of the content in any manner, without express permission of the publisher, is prohibited. No liability is assumed with respect to the use of the information herein.

Library of Congress Cataloging in Publication Data

Thompson, Mark S.  
Programming your Atari computer.

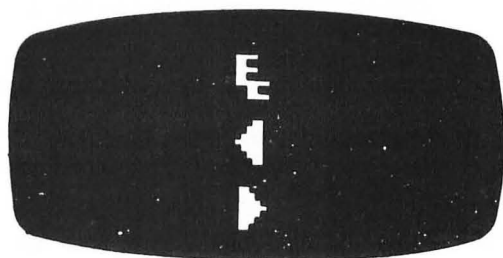
Includes index.

1. Atari 800 (Computer)—Programming. I. Title.

QA76.8.A814T46 1983 001.64'2 82-19334

ISBN 0-8306-0453-7

ISBN 0-8306-1453-2 (pbk.)



## Contents

<b>Preface</b>	<b>vii</b>
<b>1 Number Systems and Codes</b>	<b>1</b>
The Decimal Number System—The Binary Number System— Converting Decimal Numbers to Binary Numbers—Converting Bi- nary Numbers to Decimal Numbers—Octal Number System— Hexadecimal Number System—Binary Codes	
<b>2 Microcomputer Basics</b>	<b>12</b>
Computer Structure—Software—Interconnection of Elements— Scaling of Computers—Microprocessors—Computer Bus Systems—Types of Storage—Bus Control of I/O	
<b>3 Computer Arithmetic</b>	<b>31</b>
Addition and Subtraction of Binary Numbers—Multiplication and Division of Binary Numbers—Fractions and Decimals	
<b>4 Boolean Operations</b>	<b>44</b>
Boolean Algebra—Classes and Elements—Venn Diagrams— Connectives and Variables—Applications to Switching Cir- cuits—Fundamental Laws and Axioms of Boolean Algebra— Boolean Equation Simplification and Mechanization	
<b>5 Introduction to Programming</b>	<b>80</b>
BASIC—The Programming Process—Let's Write a Program— Line Numbers—BASIC Statements	
<b>6 Introduction to the ATARI 800</b>	<b>97</b>
Understanding Software—Keyboard—Program Recorder— Learning to Program the ATARI 800—Writing BASIC Source Programs—Editing Your Program—Transferring Programs to and from Cassette Tapes	

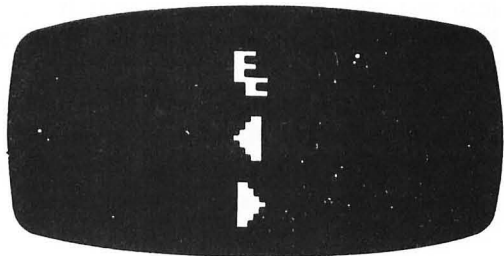


<b>7</b>	<b>Optional Peripherals and Software</b>	<b>119</b>
	8K Computer Systems—The Printer—16K Computer Systems—Controllers—The ATARI 810 Disk Drive—Disk Operating System—BASIC Commands Used with Disk Operations—Directory of BASIC Words Used with Disk Operations—BASIC Error Messages Used During Disk Operation	
<b>8</b>	<b>I/O Operations</b>	<b>135</b>
	Input/Output Devices—Input and Output Commands	
<b>9</b>	<b>Mathematical and Other Functions</b>	<b>143</b>
	Arithmetic Functions—Trigonometric Functions—Special Purpose Functions	
<b>10</b>	<b>Programming Techniques to Save Memory</b>	<b>147</b>
<b>11</b>	<b>Sample Programs: Conversions</b>	<b>149</b>
	Currency Exchange—Metric Conversions	
<b>12</b>	<b>More Programs</b>	<b>157</b>
	Finding Greatest Common Divisor of Two Numbers—Program for Gun Collectors—Checkbook Balancer	
<b>13</b>	<b>Flowcharting Techniques</b>	<b>165</b>
	Flowcharting Symbols—Control Structures—The If Statement—The Else Option—Pseudocode	
<b>14</b>	<b>ATARI Graphics and Sound</b>	<b>174</b>
	Commands to Control the Graphics—Text Modes Character Print Program—Light Show Program—United States Flag Program—Video Graffiti—Using Sound—Type-a-Tune Program—Computer Blues	
<b>15</b>	<b>Programming in Machine Language</b>	<b>198</b>
<b>16</b>	<b>Continuing Education</b>	<b>205</b>
	NRI—National Technical School—Typical Correspondence Lessons	
<b>Appendix A</b>	<b>Alphabetical Directory of BASIC Reserved Words</b>	<b>220</b>
<b>Appendix B</b>	<b>Error Messages</b>	<b>226</b>
<b>Appendix C</b>	<b>ATASCII Character Set</b>	<b>229</b>
<b>Appendix D</b>	<b>ATARI 400/800 Memory Map</b>	<b>234</b>
<b>Appendix E</b>	<b>Derived Functions</b>	<b>237</b>

<b>Appendix F</b>	<b>Printed Versions of Control Characters</b>	<b>238</b>
<b>Appendix G</b>	<b>Memory Locations</b>	<b>239</b>
<b>Appendix H</b>	<b>Glossary of Microcomputer Items</b>	<b>242</b>
<b>Index</b>		<b>270</b>







## Preface

This book is directed to the person interested in obtaining a complete understanding of the design and capabilities of the ATARI 800 computer. It opens the door to many applications of the ATARI 800. It differs from many books in that it explains fundamental microprocessor techniques while it enables you to use your own ATARI effectively.

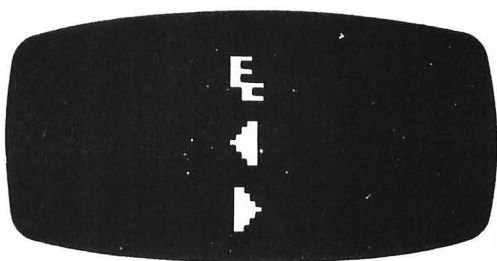
Software packages are available that will enable the user of an ATARI 800 to begin running programs without spending time learning the basic fundamentals. However, to write and run programs effectively, the programmer should have a fundamental understanding of the components that make up the ATARI 800. This knowledge begins with the microprocessor chips, memory chips, and I/O chips. It extends to the keyboard and switches for setting up an instrument or device, transducers for sensing inputs, actuators for control, devices for communicating with remote computers and other instruments, devices to provide extra storage capability, and printers and displays for informing the user of results.

Once the basic components of the machine are understood, the programmer must thoroughly understand programming processes and be able to translate these into the language of the ATARI 800. It is also helpful for the programmer to understand how the extensive requirements of the ATARI 800 can be broken down into manageable parts and how sections of programs can be written to meet each requirement. Finally, the programmer must be able to put the

software parts together into a coordinated whole so that the overall goals of the program are met.

To the novice, all of these requirements may seem a little much to conquer right at the beginning, but this book is designed to lead you step-by-step—taking first things first—until even the beginner will be able to program the ATARI 800 just like a seasoned pro.

Some of the initial data may seem immaterial at first, but as you get more and more into actual programming, all of this material will “gel”—and result in a thorough knowledge of the ATARI 800 and how to program it to suit personal needs.



## Number Systems and Codes

Binary numbers and codes are the basic language of all microprocessors—including the ATARI 800—and a good, solid foundation in these numbers and codes is essential in understanding how microprocessors operate and how they are programmed. A good knowledge of octal and hexadecimal numbers is also important as they allow easy manipulation of binary numbers and data. Understanding these systems thoroughly will give you an excellent working knowledge of the codes used in programming the ATARI 800.

### THE DECIMAL NUMBER SYSTEM

All of you are familiar with the decimal number system as this is the system you were introduced to when you first started to learn mathematics. You may remember that this system has a *base*, which indicates the number of characters or digits used to represent quantities, of 10 (the numerals zero through nine).

Other number systems also have bases, but they differ from each other. In fact, the base (sometimes called radix) is the basic distinguishing feature of all number systems. When necessary a subscript is used to show the base. To illustrate, the number  $1982_{10}$  is from a number system with a base of 10.

Each digit position in a number in the decimal number system carries a particular weight which determines the magnitude of that number. Each position has a value determined by some power of the



base, in this case 10. The value of each  $10^0$ ,  $10^1$ ,  $10^2$ , and so forth. Condensed, they are as follows:

$$\begin{aligned}10^0 &= 1 \\10^1 &= 10 \\10^2 &= 100 \\10^3 &= 1,000 \\10^4 &= 10,000 \\10^5 &= 100,000 \\10^6 &= 1,000,000 \\10^7 &= 10,000,000 \\10^8 &= 100,000,000 \\10^9 &= 1,000,000,000\end{aligned}$$

It is easy to see that for most uses, we seldom have the need to go higher than  $10^9$ —except to balance our national budget!

The small figure used to indicate the power to which a number is raised is called an exponent, and the number raised to that power is called the base number, or simply the *base*. Thus, in the previous example,  $10^0$ ,  $10^1$ ,  $10^2$ , etc., the number 10 is the base, and 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 are exponents.

The total quantity of a number is evaluated by considering the specific digits and the values of their positions. For example, the number 6403 is meaningful as written, but it may also be written in positional notation as,

$$\begin{aligned}&(6 \times 10^3) + (4 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) \\&\text{which equals,} \\&\quad (6 \times 1000) + (4 \times 100) + (0 \times 10) + (3 \times 1) \\&\text{or,} \\&\quad 6000 + 400 + 0 + 3 = 6403\end{aligned}$$

In other words, to determine the value of a number, multiply each digit by the value of its position and add the results.

In the previous examples, only *integer* or whole numbers have been discussed. Sometimes, however, it becomes necessary to express quantities in terms of fractional parts of a whole number. We must then use numbers whose positions have values that are negative powers of ten. A partial listing of negative powers of 10 are:

$$\begin{aligned}
10^{-1} &= \frac{1}{10} && \text{or } 0.1 \\
10^{-2} &= \frac{1}{100} && \text{or } 0.01 \\
10^{-3} &= \frac{1}{1000} && \text{or } 0.001 \\
10^{-4} &= \frac{1}{10,000} && \text{or } 0.0001 \\
10^{-5} &= \frac{1}{100,000} && \text{or } 0.00001 \\
10^{-6} &= \frac{1}{1,000,000} && \text{or } 0.000001
\end{aligned}$$

We could continue, but the point should be clear for our purposes.

You also know that a decimal point separates the whole and fractional parts of a number. The whole number is to the left of the decimal point and the fractional part is written to the right of the decimal point. To illustrate this, the decimal number 3.17 can be written with positional notation as follows:

$$(3 \times 10^0) + (1 \times 10^{-1}) + (7 \times 10^{-2})$$

which equals,

$$(3 \times 1) + (1 \times 1/10) + (7 \times 1/100)$$

or,

$$3 + .1 + .07 = 3.17$$

In the example above, the left-most digit ( $3 \times 10^0$ ) is the most significant digit as it obviously carries the greatest weight in determining the value of the number. The number to the far-right of the decimal point is the least significant digit as it carries the least weight in determining the value of the number.

## THE BINARY NUMBER SYSTEM

The simplest number system that uses scientific notation (exponents) is the binary number system, which consists of only two elements, and is expressed as a base of 2 (0 and 1). These two digits have the same basic value as 0 and 1 in the decimal number system.

Because of its simplicity, microprocessors use the binary number system to manipulate data, which is represented by binary digits called bits. The term "bit" is derived from the contraction of

**Table 1-1. Decimal and Binary Positional Values.**

Decimal	$10^3$ 1000	$10^2$ 100	$10^1$ 10	$10^0$ 1
Binary	$2^3$	$2^2$	$2^1$	$2^0$
Value expressed in decimal	8	4	2	1
Value expressed 1000 in binary		100	10	1

*binary digit*. Microprocessors operate on groups of bits which are referred to as words. The binary number 00001101 contains eight bits.

When dealing with the binary number system, it is very important not to confuse the numbers involved. The binary number 10, for example, should be thought of being “one, zero” and not “ten” as it would in the decimal number system. Ten has no meaning in binary notation. Similarly, the binary number 100 should be thought of as “one, zero, zero” and not as “one hundred.” Always think of the binary numbers as “ones” and “zeros.”

Note that when you are dealing with several number systems, it is best to identify a number with an appropriate subscript. For example, the notation  $10_{10}$  identifies the number as being in the decimal system (base of 10) and the number is “ten.” The notation  $100_{10}$  would be “one hundred.” The notation  $100_2$  is “one, zero, zero” in the binary system;  $100_8$  is “one, zero, zero” in the octal system; and  $100_{16}$  is “one, zero, zero” in the hexadecimal system. The latter two systems will be discussed later in this chapter.

In the binary system, you will either multiply  $2^0$  by one or by zero, as we are using only two numbers. The positional values in the decimal and binary systems are shown in Table 1-1. Notice that the power to which the base is raised is the same in both systems. To better understand these positional values, let’s express decimal 1 in binary numbers. Binary  $2^0 = 1$ , and one times 1 equals one. Therefore, decimal 1 equals binary 1.

Decimal 2 expressed in binary numbers means that we move over into the next position in Table 1-1 and find that by multiplying  $2^1$  by one results in 10 or “one, zero” equals decimal 2.

## CONVERTING DECIMAL NUMBERS TO BINARY NUMBERS

In programming the ATARI 800 and other microprocessors,



you will often need to determine the decimal value of binary numbers. In addition, you will find it necessary to convert a specific decimal number into its binary equivalent.

The chart in Table 1-2 gives the decimal/binary equivalents for numbers from 0 to 16. This would be very helpful if they were the only numbers encountered in programming. Actually, this is not the case; you will need to convert many different numbers and an easier method must be used. Since a binary number is based on powers of 2, we can convert a decimal number to binary by repeatedly dividing by two.

To simplify matters, let's take the number  $10_{10}$ . Set up your division as follows:

2

|

10

2

|

5

2

|

2

2

|

1

0

0

1

1010

Notice we first take the number 10 and divide by 2. Two goes into 10 five times with a remainder of zero. We write the 5 beneath the 10 and the 0 to the right of the vertical line. Now divide 2 into 5, it goes two times with a remainder of one. Write the 2 beneath the 5 and the remainder 1 beneath the 0 to the right of the vertical line. Divide 2

Table 1-2. Decimal/  
Binary Equivalents.

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000

into 2; it goes 1 time with a remainder of 0. Write the 1 beneath the 2 to the right of the vertical line. Finally, divide 2 into 1; it goes 0 times with a remainder of 1, so write the remainder to the right of the vertical line. The digits to the right of the vertical line are the binary equivalent of the decimal number. The number is written from right to left working from top to bottom, or from left to right working from bottom to top. Therefore, the binary equivalent of 10 is 1010, as listed in Table 1-2.

The same procedure can be used no matter how large the numbers. Just continually divide by 2, placing either a 0 or a 1 to the right of the vertical line. If you haven't tried converting decimal numbers to binary, work a few examples from numbers, say, 12 to 16 and then check your answers with Table 1-2.

Example of this type of method (using fractions) is shown in Table 1-3. The decimal fraction 0.90625 is continually multiplied by 2! Did you say multiply? Yes, to convert a decimal fraction to a binary number, multiply the fraction successively by the desired base (two in our case) and record any integers produced by the multiplication as an overflow. These calculations will result in numbers with a 1 or 0 in the units position. Subtract the unit and multiply the rest by 2 again. The decimal fraction 0.90625 is converted as follows:

**Table 1-3. Converting Decimal Fractions to Binary. Note That the Original Number Is First Multiplied by 2, the Product of which is Multiplied by 2, and so on Until the Number Comes Out Even. There Will Be Times, However, Where the Number May Not Come Out Even—Just Like in the Decimal System. But All Can Be Carried Out for Practical Accuracy.**

0.90625	×	2	=	1.8125	=	0.8125 with overflow	1
0.81250	×	2	=	1.6250	=	0.6250	1
0.6250	×	2	=	1.2500	=	0.2500	1
0.25000	×	2	=	0.5000	=	0.5000	0
0.50000	×	2	=	1.0000	=	0	1

Notice that the multiplication process is continued until either 0 or the desired precision is obtained. The overflows are then collected beginning with the most significant value, which is the first digit to the right of the binary point, and proceeding to the least significant value. The number is  $0.11101_2$ .

If the decimal number contains both an integer and fraction, you must separate the integer and fraction using the decimal point as the break point. Then perform the appropriate conversion process on each portion. After you convert the binary integer and binary fraction, recombine them.

CONVERTING BINARY NUMBERS TO DECIMAL NUMBERS

The easiest way (and the way recommended initially) to convert binary fractions to decimal numbers is to use a table, such as the one shown in Table 1-4. To use this table, let's take a number, say binary 1011010 and convert it to its decimal equivalent. There are seven digits in the number 1011010. The first 1 (the digit farthest left) is in the  $2^6$  column and is equal to the decimal number 64. There is a zero under  $2^5$ , so there is nothing taken from this column. Under  $2^4$  there is a 1, so we write the value 16 next to our previous number 64. There is a 1 in the  $2^3$  column, so an 8 is placed next to the 16. In the  $2^2$  column there is a 0, and therefore, the value will be omitted. But there is a 1 beneath the  $2^1$  column and this means a 2 is placed next to the previous 8. There is a 0 beneath  $2^0$ , so we omit this value also. Now the numbers 64, 16, 8 and 2 are totalled for a sum of 90—the decimal number equivalent to 1011010.

Refer again to the table in Table 1-4 and follow the description of the conversion step-by-step until you are certain that you understand how the conversion was made. Then practice some conversions of your own. Check your answers by reversing the conversion process; that is, use repeated divisions (or multiplications, depending upon the case) by 2.

OCTAL NUMBER SYSTEM

Octal is another number system, this time with a base of 8. It uses the digits 0 through 7, and these have the same basic value as the digits 0 through 7 in the decimal number system. Actually, octal numbers are really nothing more than 3-bit groups of binary numbers. In fact, octal numbers are a form of binary shorthand.

To convert a binary number to an octal number, first write down the binary value and, starting from the right-hand end of the

Table 1-4. Table for Converting Binary Fractions to Decimal Numbers.

$2^{-1}$ 0.5	$2^{-2}$ 0.25	$2^{-3}$ 0.125	$2^{-4}$ 0.0625	$2^{-5}$ 0.03125	$2^{-6}$ 0.015625
$2^{-7}$ 0.0078125	$2^{-8}$ 0.00390625	$2^{-9}$ 0.001953125	$2^{-10}$ 0.0009765625		

**Table 1-5. Binary-to-Octal Conversion.**

Octal	Binary					
0	000					
1	001					
2	010					
3	011					
4	100	01	010	110	011	(binary)
5	101	1	2	6	3	(octal)
6	110					
7	111					

number, mark off groups of three bits. Now, using Table 1-5, write down the equivalent octal value in each group. Notice that this conversion is exactly like binary-to-decimal conversion, except that the decimal digits 8 and 9 don't exist in the octal system.

Converting a decimal number to octal is accomplished in the same manner as decimal to binary except that the base number is now 8 rather than 2. Therefore, write the decimal number and divide by 8. For example, to convert the decimal number 469 to octal, set up your division as shown in Fig. 1-1. Note that the decimal number 469 is equal to 725 octal number. Just keep dividing the decimal number by 8 until 0 results.

To convert from an octal number back to a decimal number, we use the same procedure as we used to convert a binary number back to a decimal number. In the binary conversion, a table showing different powers of 2 was used. In the octal system, a table based on the values of powers of 8, as might be expected, is used. Decimal-octal equivalents are shown in Table 1-6.

## HEXADECIMAL NUMBER SYSTEM

Hexadecimal ("hex") is another number system that is often used with microcomputers. It is similar in value structure to the

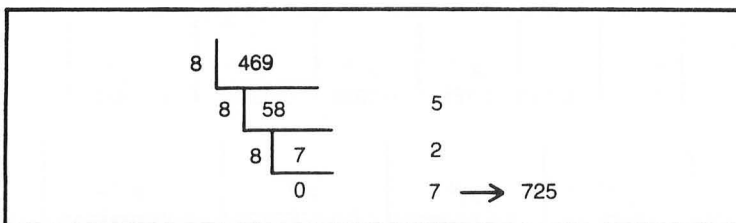


Fig. 1-1. Example of converting decimal number to the octal number system.

**Table 1-6. Decimal/Octal Equivalents.**

Decimal	Octal	Decimal	Octal
0	0	10	12
1	1	11	13
2	2	12	14
3	3	13	15
4	4	14	16
5	5	15	17
6	6	16	20
7	7	17	21
8	10	18	22
9	11	19	23

Decimal	Octal
20	24
21	25
22	26
23	27
24	30
25	31
26	32
27	33
28	34
29	35

octal number system, and allows easy conversion with the binary number system.

As might be imagined, the base is 16 for the hexadecimal system. The symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In hex, A, B, C, D, E, F, are digits, not letters; that is, just as the next digit after 8 ( $8 + 1$ ) is 9, the next digit after 9 ( $9 + 1$ ) is A, the next digit ( $A + 1$ ) is B, and so forth.

Conversion of binary numbers to hex is similar to converting binary numbers to octal except that 4-bit groups are used instead. The conversions are shown in Table 1-7.

You can convert from a decimal number to a hex number the same way you did from a decimal number to an octal number, except that you divide by 16 instead of 8. In converting from decimal to hex, it's recommended that you first convert the decimal number to binary, and then convert the binary value to hexadecimal. Similarly, to convert from hexadecimal to decimal, first convert from hex to binary and then go from binary to decimal.

To illustrate, to convert decimal 685 to hex, convert decimal 685 to binary, which will be 0010 1010 1101. Mark the numbers off in groups of four bits from the right; that is,

$$\begin{array}{r|l|l} 10 & 1010 & 1101 \\ 2 & A & D \end{array}$$

and the result is 2AD in hex.

### BINARY CODES

Has the converting of decimal number to binary or binary shorthand systems reminded you of anything? Sure, it's obviously a form of code just like you see in the cloak and dagger films. The act of conversion is called "coding" and codes and coding is what this section is all about.

The decimal number system is easy to use because it has more than likely been the system you have used most in your life. The binary number system is less convenient to use because it is less familiar, and few people can glance at a binary number and recognize its decimal equivalent. You can readily calculate its value within a few minutes, but you probably won't recognize it at a glance. Therefore, the amount of time it takes to convert or recognize a binary number quantity is a distinct disadvantage in working with this code despite the numerous hardware advantages. Computer designers have therefore come up with a special form of binary code that is more compatible with the decimal system. This particular code is known as *binary coded decimal*, which combines some of the

Table 1-7. Binary-to-Hex Conversion.

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
010 2	1011 B	011 3
		(binary) (hexadecimal)

**Table 1-8. Standard 8421 BCD Code and the Decimal Equivalents.**

Decimal	8421 BCD	Binary
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	0001 0000	1010
11	0001 0001	1011
12	0001 0010	1100
13	0001 0011	1101
14	0001 0100	1110
15	0001 0101	1111

characteristics of both the binary and decimal number systems.

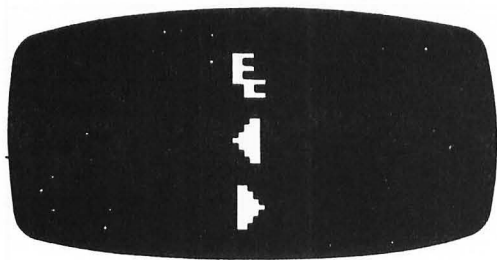
In general, the BCD code is a system of representing the decimal digits 0 through 9 with a four-bit binary code, using the standard 8421 position weighting system of the pure binary code. The standard 8421 BCD code and the decimal equivalents are shown in Table 1-8. Converting the BCD numbers into their decimal equivalents is done by simply adding together the values of the bit positions whereby the binary 1's occur. Note, however, that there are only ten possible valid 4-bit code arrangements. The 4-bit binary numbers representing the decimal numbers 10 through 15 are invalid in the BCD system. To represent a decimal number in BCD notation, substitute the appropriate 4-bit code for each decimal digit.

One big advantage of the BCD code is that the ten BCD code combinations are easy to remember once you have worked with them a while. In fact, after some time spent in learning the BCD code combinations, you should be able to make the conversion almost as quickly as if the number were already in decimal form!

One big disadvantage of the BCD code is that it is less efficient than the pure binary code because it takes more bits to represent a given decimal number in BCD than it does using pure binary notation.

There are other codes used in microprocessors, but for now, we will stop at pure binary and the BCD numbering system.

## Chapter 2



### Microcomputer Basics

A microcomputer is a very complex electronic device consisting of thousands of microscopic transistors squeezed onto a tiny chip of silicon, related leads, keyboard, housing, and other mechanical and electronic components. Regardless of the type of microcomputer used, the programmer should have a basic knowledge of how these devices operate.

In general, there are two basic types of computers: digital and analog. Digital computers represent everything in terms of digits. That is, all data is discrete and discontinuous. There are some hybrid computers that connect digital computers to analog computers, but most of today's computers are of the purely digital kind. You should now be getting an idea of why certain number systems were covered in Chapter 1.

An analog computer uses a quantity, such as an electrical voltage, as an analog of some other quantity. For example, 3.145 volts in an electrical circuit might represent a velocity of 3,145 feet per second in a ballistics simulation program. Analog computers are fast for certain specialized applications, but they are not practical for many purposes.

To help you understand the essential elements of a digital computer, well-known analogy will be presented. Imagine a faithful, loyal, dedicated employee, such as a secretary, who follows every instruction to a "T." This secretary, given a manual of procedures and some documents to work with, can refer to history



files and produce required reports for your company. He or she needs only a calculator (to perform computations) and a sheet of instructions. If he unerringly follows those instructions, he will produce a stack of papers in his out-basket that represents a transformed version of the information contained on papers in the in-basket. If the secretary transforms legitimate inputs into correct outputs by following the instruction sheet, those instructions can be considered a correct and valid "program."

In the example above, the secretary performs five distinct functions.

1. Reading source information (both the documents in the in-basket, and the list of instructions).
2. Storing intermediate results (remembers information for a short time, either in her head or on a scratch pad).
3. Computing, using a desk-top calculator (to find totals or to arrange data into proper order).
4. Writing transformed information (the required results in the out-basket or records in files for future use).
5. Controlling these activities (sequencing of these four activities to achieve the required result).

Figure 2-1 shows that a computer can perform these five functions, but all of the data and instructions must be supplied in a special computer-readable form. In other words, the *program* is a computer's equivalent of a manual of procedures. Under the guidance of the steps in that program, the computer may read input data, store intermediate results, perform arithmetic using numbers from inputs and from previous computations, and write outputs in many forms. The control element of the computer selects which of the other four elements are to be used at any given time. The computer, then, performs five functions similar to those performed by a secretary:

SECRETARY	COMPUTER
Read	Input
Store	Storage
Compute	Arithmetic
Write	Output
Control	Control

As human beings, we perform input functions through our senses: taste, smell, touch, sight, and hearing. Some computers

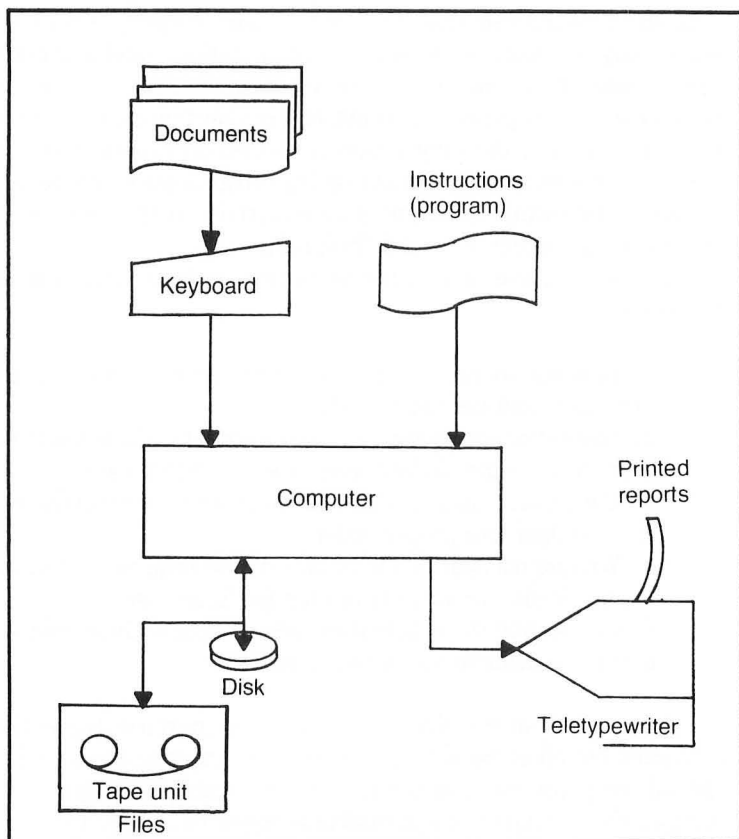


Fig. 2-1. A computer performs functions that are similar to those performed by a secretary.

sense inputs by sight (optical character readers, for example) or feel (wire contacts through holes in punched paper tape). Some inputs are simple electrical on-off signals from remote sources. Sound also plays an important part in allowing computers to read.

Humans store information for short-term retention in their memory, relying on a variety of filing systems for more permanent storage. Computers also have memory. Their capacity to store information varies from machine to machine, depending on the number of external devices, such as tapes, disks, and drums, that are used. The storage capacity of computers using these devices is almost infinite.

Both man and his machines can perform arithmetic. The arithmetic capability of digital computers ranges from elementary to

complex. As you learned in an earlier chapter, computers perform all arithmetic operations using binary numbers. The computer can also perform other “logical” functions such as determining whether or not one number is greater than another.

We create output by our actions. We write, speak, or change a situation appropriately. The computer produces output in the form of rewritten magnetic tapes, disks, and drums; printed reports; and even motor on/off controls. Since the computer operates electronically, it can also control electronic circuitry, such as the circuitry necessary to automatically open or close a flow valve in a water supply system.

We can control both ourselves and our machines. Machines can control only what they are instructed and permitted to control. When the secretary does some work, he or she is responsible for controlling the order in which various steps of the procedure are performed. He can, under the guidance of the detailed instruction list, read some information, perform some computations, and produce some output in the order specified in the program.

Computers are controlled by computer programs. The program is a description of a detailed procedure that is to be followed. Just as a theater program outlines the events that are to follow, a computer program outlines the steps to be followed to arrive at a solution to the problem presented. The program is interpreted by the computer which turns various circuits on and off in the sequence necessary to produce the solution.

## COMPUTER STRUCTURE

The diagram in Fig. 2-2 shows the five basic elements of a digital computer. Every digital computer, no matter how large or small, has these five elements plus the set of instructions that tells the computer what to do. A computer must be able to accept input data, process it through arithmetic or logical operations, store some intermediate results, and eventually produce output data—all under the guidance of a central control element. The control element is guided in its operations by the instructions supplied by a programmer. The five basic elements comprise the hardware of the computer. Hardware can be touched, squeezed, observed; it is tangible. The instructions, on the other hand, represent the software of a computer. We may be able to observe, touch, or squeeze a storage medium, but the specific bit patterns held within are not physical; they are information.

**Inputs:** Nearly all computers require inputs from one or more

external sources. A computer that receives no input signals can only repeat the same sequence of instructions over and over again. Some small computers are used in precisely this way, but most computers have inputs that can be brought into the computer for subsequent processing.

To compute a payroll check, for example, the computer must have the ability to fetch an employee record from a file and to read in the number of hours worked in the week. These both require an input path to the computer. In some simple applications where small computers are used to replace older electromechanical logic, inputs may not be required. A programmable controller of a numerically-controlled milling machine may not require inputs. The program contains the sequence of instructions which cause the machine to mill the same part repeatedly. External inputs are of no value in this case.

Inputs to the computer may take the form of punched cards, magnetic tapes or disks, or simple switch closures. When a keyboard is operated by a human, each key depression results in a switch closure. The computer can accept this switch closure as a binary signal and convert the key depression into a binary pattern.

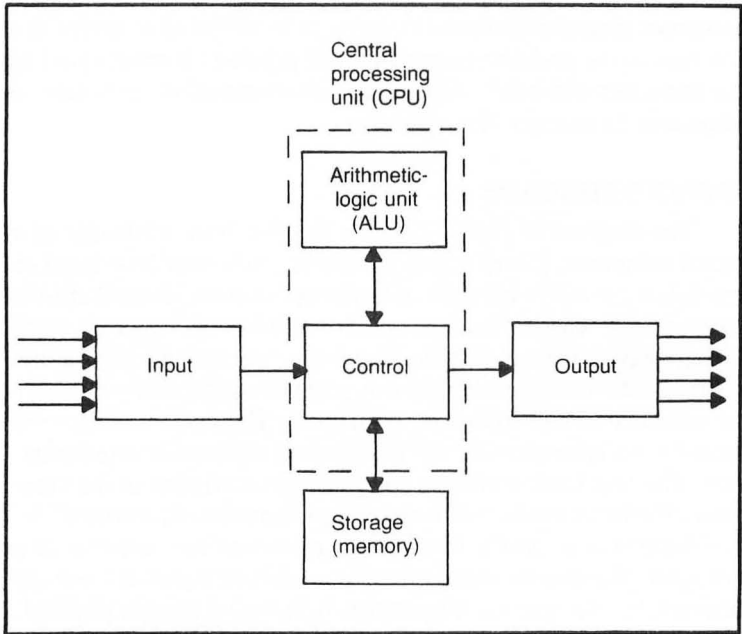


Fig. 2-2. Five elements in all digital computers.

That binary pattern, in turn, can be used to print a selected character on a printer or display it on a cathode-ray tube (CRT).

**Outputs:** A computer must produce outputs to be useful. A computer's output may be in the form of printed results, graphic plots drawn with special plotting equipment, or simply an electrical signal that can turn a lamp on or off. The range of possible outputs from a computer is virtually unlimited. The output element of a computer is selected when the control unit executes an instruction that transfers data from the arithmetic-and-logic unit (ALU) storage area, or when an input is to be output. Most often, the data comes from storage, and was created by a complex combination of inputs that were modified by the ALU.

The input and output devices are points at which humans interact with the computer. Much care must be taken to make computers easy for people to use, rather than making people adapt to poorly designed or poorly programmed computers.

**Storage:** In most computer applications, data must be stored before it is processed completely or used to produce outputs. The electrical digital signals that represent the data values are placed in the storage element, or memory. Instructions can also be read into storage and held for future use.

**Arithmetic:** A computer processes information. That processing is performed by the ALU which accepts data from input and storage elements, and produces computed results for storage or output. To add a pair of numbers, the ALU is directed to fetch the values from storage, and to place the sum back into storage. The ALU may include capabilities for addition, subtraction, comparison, logical operations, and even multiplication and division.

**Control:** All elements of a computer operate under the direction of the control element which opens and closes the various pathways for data from input to storage, from storage to the ALU, and from the ALU to output. The control element operates as directed by the instructions provided in storage. By writing the proper sequence of instructions, the computer programmer can completely control the detailed transfers of data within the computer to produce desired results. Thus, with different programs, the computer can be made to perform different functions, even though the underlying electronic circuitry remains unchanged.

## SOFTWARE

Software consists of a collection of instructions that can be followed by the control element of the computer. These instruc-

tions must be detailed step-by-step instructions. Unlike humans, who can inject common sense into the completion of a task, a computer is not capable of logical reasoning. A computer must be ordered to do even the simplest part of a task. There are four major classes of computer instructions:

1. Input/output instructions cause data to be accepted as inputs, or to be produced as outputs.
2. Arithmetic-logic instructions cause data to be transformed through simple operations in the ALU. These operations include addition, subtraction, shifting, and logical operations (AND, OR, etc.).
3. Storage instructions move data between storage and other elements.
4. Control instructions specify changes in the normal sequence of instruction execution.

The *instruction set* is the collection of general-purpose instructions available in a given computer. This set of instructions can show you how that computer works. The number and type of instructions are an indication of the ease with which the computer can perform operations. Some computers, for example, have a multiply instruction. Given two numbers, this instruction will produce the product. Most microcomputers, on the other hand, do not have a multiply instruction. Multiplication can be done on these computers, but many more instructions must be executed. Since most microcomputers are used in applications that do not require multiplication, the lack of a multiply instruction is seldom a problem. However, it is important to note that those facilities that are not provided in the instruction set can be created through the execution of the right sequence of simpler instructions.

## **INTERCONNECTION OF THE ELEMENTS**

Each element of a computer can be treated as a subsystem. Each element has certain inputs and outputs, and certain actions that take place internally to transform those inputs into outputs. A revised view of the five-block model of the computer is shown in Fig. 2-3. Here, all of the inputs and outputs to each element of the computer are identified. Note that except for the interfaces be-

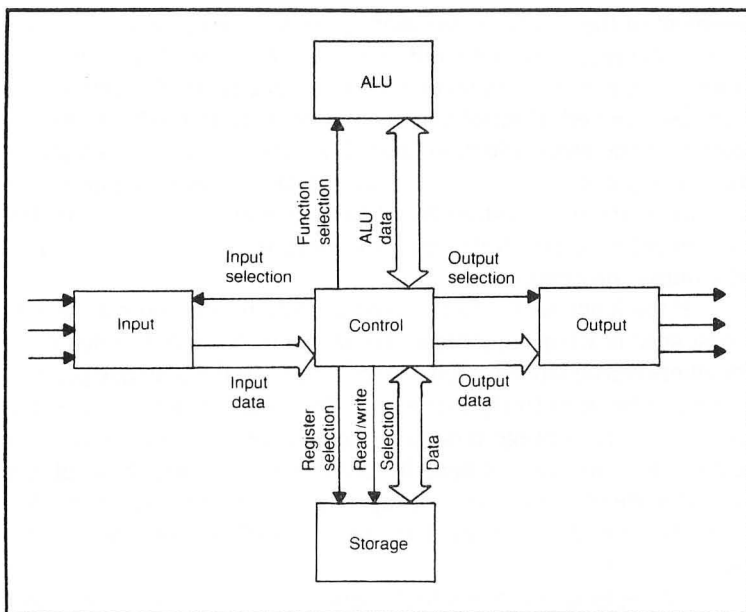


Fig. 2-3. The interconnections between elements consist of both data and control signals.

tween the computer and the outside world, the output from one element becomes the input for another element. By analyzing how these paths between elements operate, we can discover how a computer functions in detail.

Most of the pathways for data are bidirectional. However, issuing data to an input source, or attempting to acquire data from an output destination is meaningless. These data pathways can be easily limited to a single direction. On the other hand, data must move in both directions between control and storage, and between control and the ALU. Since data may flow both ways, these paths are called bidirectional.

Notice that all of the control signals originate in the control element, and that all data passes through the control element. "Control" may be thought of as a gigantic switch-board that operates at microsecond rates to switch data from one source to another destination. In fact, this is precisely how contemporary computers work.

At any one time, control may be directing one of the other four elements to perform some operation. In very complex computers, more than one element may be accessed at one time. The input

element of the computer has many possible data sources, but only one of them is to be selected at a time. When reading from input sources, the control element must first specify which input source is to be selected. Control places some bit pattern on the wires that comprise the input selection lines. The selected data is then placed on the input data lines. Once input data has been acquired, the control element must store that data somewhere for future use. The data might be shipped off to the ALU, or to storage, or even to one of the output destinations.

If the input data is to be saved in storage, the control element must specify where in storage the data is to be placed. Since many numbers can be stored, each one must be stored in a unique place so each can be selectively retrieved. The data is placed on the data lines, and the storage module is commanded to *write* (store) the data. The command is sent by the control element through the Read/Write selection line. The specific register into which the data is to be stored is specified by a pattern of bits on the register selection lines.

The specification of which input lines to read and into which register to place the data in storage, is carried in an instruction. That instruction contains three essential items of information:

1. The operation to be performed (read data from an input and store it away).
2. The source of the data (the specific input source to select).
3. The destination of the data (the specific register into which the data is to be placed).

One way to visualize this operation is to examine a small sequence of instructions that might be found in a large program. This brief instruction sequence (program segment) acquires two input numbers and produces a sum which is output. This program is summarized in the following sequence of instructions.

1. Accept a number from input source 5, and save that number in storage at register 83.
2. Accept a number from input source 2, and save that number in storage at register 43.
3. Read the first number (from register 83) and pass it to the ALU, commanding the ALU to save that number.
4. Read the other number (from register 43) and pass it to the



ALU, commanding the ALU to add it to the previously saved number.

5. Accept the sum from the ALU and pass it to output destination 7.

Each of these instructions can be described in more detail by referring to the signals that appear on the wires between the various computer elements. For example, the first step above (accept a number and place it in storage) can be described in more detail like this:

1. Control issues the number of the required input source (5) on the input selection lines. The input element accepts that number and selects that data source.
2. The data source presents some data (say, the number 113) on the input data lines to control. Control accepts that number and holds it temporarily.
3. Control issues the storage register number (83) to storage by placing that number on the register selection lines. Control also commands the writing action on the Read/Write selection line. Storage accepts the register number and gains access to that register.
4. Control places the temporarily held data (113) on the data lines. Storage writes that number into the selected register.

Obviously, such detailed English explanations could be difficult to follow for any reasonably complex set of instructions. Therefore, a shorthand method is sometimes used to describe those sequences. The narrative description for reading and storing a number could be written in a briefer form like this:

1. Input selection control (5);  
Input input selection (5)
2. Input data input (113, from source 5);  
Temporary input data (113)
3. Register selection control (83);  
Read/Write selection control (Write);  
Storage register selection (83)
4. Data temporary (113);  
Storage (at register 83) data

Each of the items named in this brief example is a register in the computer. One of the control registers (which happens, in this example, to contain the number 5) supplies its contents to the input selection register.

Now, reconsider the application posed earlier: the production of the sum of two numbers. In the following example, each of the original instructions has been broken down into a number of detailed steps. These steps are represented in the shorthand form. You should fill in the descriptive shorthand for the last instruction yourself to be certain that you understand exactly how the computer works:

1. Input selection control (5);  
Input input selection  
Input data input (113);  
Temporary input data  
Register selection control (83);  
Read/Write selection control (Write);  
Storage register selection  
Data temporary;  
Storage register 83 data (113)
2. Input selection control (2);  
Input input selection  
Input data input (102)  
Temporary input data  
Register selection control (43);  
Read/Write selection control (Write);  
Storage register selection  
Data temporary;  
Storage register 43 data (102)
3. Register selection control (83);  
Read/Write selection control (Read);  
Storage register selection  
Data storage (113);  
Temporary data  
Function selection control (Save);  
ALU data temporary (113)
4. Register selection control (43);  
Read/Write selection control (Read);

Storage register selection  
Data storage (102);  
Temporary data  
Function selection control (Add);  
ALU data temporary (102)

5. Function selection \_\_\_\_ (Fetch);  
Temporary \_\_\_\_  
Output selection \_\_\_\_ (\_\_\_\_);  
\_\_\_\_ output selection  
\_\_\_\_ \_\_\_\_;  
Output \_\_\_\_ (215)

## SCALING OF COMPUTERS

It might appear that the size of numbers is limited by the widths (number of data lines) of the data paths that exist between elements. Actually, in the example, data path widths of eight bits would be sufficient, since the numbers never got larger than 215 (an 8-bit register can hold numbers up to decimal 255). However, even larger numbers can be handled by breaking them up into pieces. Then each part of the number has to be handled independently, and that would require more instructions.

Large computers have wide data paths; smaller computers have narrower paths. Some of the largest computers have paths as wide as 64 bits for numbers. Programmers seldom have to resort to the slow technique of breaking a number up into pieces and processing each piece separately. On the other hand, a small microprocessor may have paths as narrow as four bits. Processing a 64-bit number would take at least 16 times as long as it would take on the larger data path.

There are special electronics techniques that can be used with large computers to make them operate much, much faster. However, because these techniques are only economical with very large computers, they are seldom used with microprocessors. Some of the largest computers can execute instructions 400 times faster than a typical microprocessor. That speed advantage, multiplied by the speed advantage of wide data paths, marks the difference between large, medium, and small computers.

Large-scale computers, or supercomputers, have thousands of ICs and require huge rooms for installation. Purchasing one hour of computer time may cost more than \$3000. The only way these fast, powerful computers can be economical is to have a large number of

users. Each user occupies only a small percentage of the computer resources, and for only a short period of time.

These large computers have large input/output systems with millions (or even trillions) of bits of storage on magnetic tapes and disks. The main storage module of the computer may have over one million registers, and the ALU-control element may be designed to permit hundreds of people to use the computer simultaneously. Most of these people use supercomputers to solve mathematical and engineering problems. These kinds of computers are seldom used for commercial data processing.

Commercial data processing is most often done on a computer scaled to the size of the business. For example, large insurance firms use complex medium-scale computers. Accessing information about policies and billing data requires a large input/output capacity, but the need for processing is relatively small. However, a relatively large computer is required to support hundreds of computer terminals in agents' offices throughout the country.

A typical medium-scale computer might have disk and tape storage that rivals that are used with a supercomputer for capacity. However, the central processing unit (CPU) will seldom be as large or complex, and storage may be limited to 250,000 registers or so. An hour of computer time on such a system might cost \$800. By using  $\frac{1}{4}$  of the computer's capacity for three minutes, a business executive can perform all the computations required for a small office at a cost of about \$10.

Small-scale computers are usually called minicomputers. Some minicomputers are used in small- to medium-sized organizations for business data processing. However, more of them are used to perform communication chores for a larger computer. The minicomputer may service hundreds of telephone lines on the "front-end" of the computer, passing data to the central medium-sized computer over a set of highspeed wires. Other minicomputers are used to effect on-line control of production processes.

A typical minicomputer might have 65,000 registers of main storage, and a relatively simple ALU-control element. Some applications might demand disk or tape storage, but many do not. The operating cost of a minicomputer might be as low as \$40.00 an hour. By occupying the entire computer for 15 minutes, an engineer may be able to control an experimental process at a cost of about \$10.

The least expensive computers are microcomputers. Some of these are used in small businesses as data processing tools; others

are used for personal entertainment; and still others are installed inside other equipment for internal control. These computers are so inexpensive that dedicating one to a single application is feasible. A small retail establishment might use a microcomputer with disk storage, 16,000 bytes registers of main storage, and a simple ALU-control element. The cost of operating such a microcomputer system might be as low as \$10 a day. By having the entire computer available at all times, the retailer may be able to better control store operations at a cost of about \$10 a day.

Technologically, all of these computer alternatives are alike except for their underlying capacity. The lower capacity computers cost less, but take longer to perform their tasks. Finding an account balance for a customer in a bank might require 1/10,000 of a second on a large supercomputer and two seconds on a microcomputer. Will the customer notice? Is the faster speed necessary, especially in view of the fact that the teller's terminal may require 20 seconds to print out the account information? In other words, for many applications, time is not a critical issue, and a less expensive computer can be used.

## **MICROPROCESSORS**

When the CPU (the ALU and the control element) of a computer is implemented in a single integrated circuit (IC), we call that the central processor a *microprocessor*. The term *microprocessor* is a contraction of microelectronics and central processor, and indicates the technology used in the heart of the computer.

A microprocessor is not an entire computer: it is only the ALU and the control portion of a computer. When the microprocessor is combined with storage, and input/output devices, however, the result may be properly called a microcomputer. A microcomputer is any computer based on a microprocessor.

At this point, it is important to distinguish between two different kinds of microcomputers. One kind integrates all computer elements on one IC or on a very small number of ICs. A "one-chip microcomputer" cannot be distinguished from its microprocessor counterpart by simple inspection. However, one-chip microcomputers are small, powerful computers that are useful for a wide range of special applications. They are found in consumer goods, electronic games, and electronic interfaces between larger computers and peripheral devices like cassette tape drives.

One-chip microcomputers are inexpensive, but limited. They

are useful in certain low-cost applications that demand little in the way of computing power or storage capacity. However, because the pins of the IC are devoted to input and output wires, these devices have little expansion capability.

The more popular kind of microcomputer is made up of separate ICs for the CPU, storage, input, and output. All these elements are interconnected by wiring etched on a printed circuit board. When a microcomputer is created out of many individual elements, as are many of the popular single-board computers, more power is available to the user. If the application demands more memory, more memory boards can be added to the configuration. If special I/O capabilities are required, they can be installed as special boards. Although more expensive than the one-chip microcomputers, microcomputers in printed circuit board form may be used in vastly more complex configurations, and may contain an almost limitless variety of storage and I/O capabilities.

Many of the general-purpose single-board computers use many one-chip microcomputers to act as I/O interfaces. A single computer complex might have dozens of one-chip microcomputers performing peripheral interface chores under internal software control—all supporting a single fast microprocessor with a large amount of storage on other boards. More and more interface systems are composed of one-chip microcomputers that have been suitably programmed. Most of these can be treated, however, as dedicated ICs. Internally, they've been programmed to perform some specific task. However, since the user or maintenance technician cannot change any of that programming, the programmed one-chip microcomputer is effectively a special-purpose IC developed for the intended interface application. The trend is away from huge centralized computer systems to systems of computers in which the computing power is distributed among many small one-chip microcomputers.

## **COMPUTER BUS SYSTEMS**

A computer *bus* can be defined as a group of wires that allow storage, control, and I/O devices to exchange information. One of the best ways to achieve high-speed operation in a computer is to keep all of the buses or pathways between various computer elements separated. In addition to the pathways to and from control as shown in Fig. 2-3, very fast computers include special pathways between other elements. A special circuit path called direct memory access (DMA) may exist between input and storage elements so

that data can be sent rapidly to storage while other operations are being performed. The more complex the computer, the more likely it is to have some special interelement paths. However, most microcomputers are designed with only simple pathways. Efficiency is sacrificed for the simplicity of a more organized structure.

## Master Bus Control

The bus system can be used to transfer information between a *master* (controlling) and a *slave* (controlled) element of the computer. The design of the master element, normally what has been previously referred to as “control,” defines the order, arrangement, and timing of the various electrical signals that affect information transfer. All slave elements must comply with the rules laid down by the master element.

Data is always transferred between one of the slaves and the master that is in control of the bus system. Since there is only one address (or location) on the *address bus* (shown in Fig. 2-3 as the “register selection” pathway) at a time, only a source or a destination register can be specified. One of the bits on the *control bus* (the path linking control to storage labelled “Read/Write selection” in Fig. 2-3) specifies whether the master is the source (and, hence, the slave is the destination) or the destination (in which case data is being read from the slave). Of course, with two separate address buses, data could be transferred directly between two slaves under master element control. However, the improvements in information transfer rates that can be obtained by having more than one address bus are generally not worth the effort on small computers. Such multi-bus systems are found on large computer systems where speed is paramount.

## The Control Bus

The key to understanding a computer is to examine its control bus. The address and data buses of a computer may be wider, but the control bus is always the most complex of the three. The address bus is always used to select one particular register to be involved in a transfer to or from the microprocessor. The *data bus* is always used to transfer data. But the control bus is composed of many different kinds of signals, all of which complex interpretations over time.

## Writing to Storage

Writing or storing data takes place in a fashion analogous to

reading or retrieving data. The data is put on the data bus by the CPU. (For those who are curious, Fig. 2-4 shows a detailed view of the address bus. VMA and /W (low) signals are produced just like the equivalent signals used during reading.) However, the storage element cannot take the data from the data bus until the specific register has been addressed. It takes time to gain access to the required register. This is called the access time. (In most computers with a sequence of this kind, the storage element would cause writing to occur at the instant the E signal made the end-of-cycle high-to-low transition.)

Transferring data from one storage register to another simply requires two successive read-and-write storage cycles. Some instructions in the computer cause successive cycles like this to happen automatically. Invariably, however, the process ends with another storage read operation which will retrieve another instruction from the program that is in memory for execution.

### Master Clocks

To create the enable (preparation) signal for controlling the

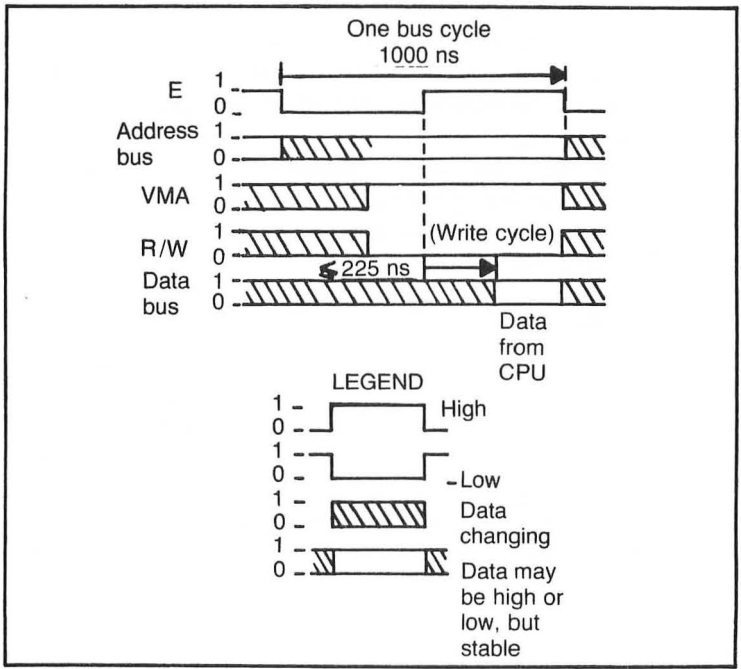


Fig. 2-4. A typical timing diagram for a write to storage.



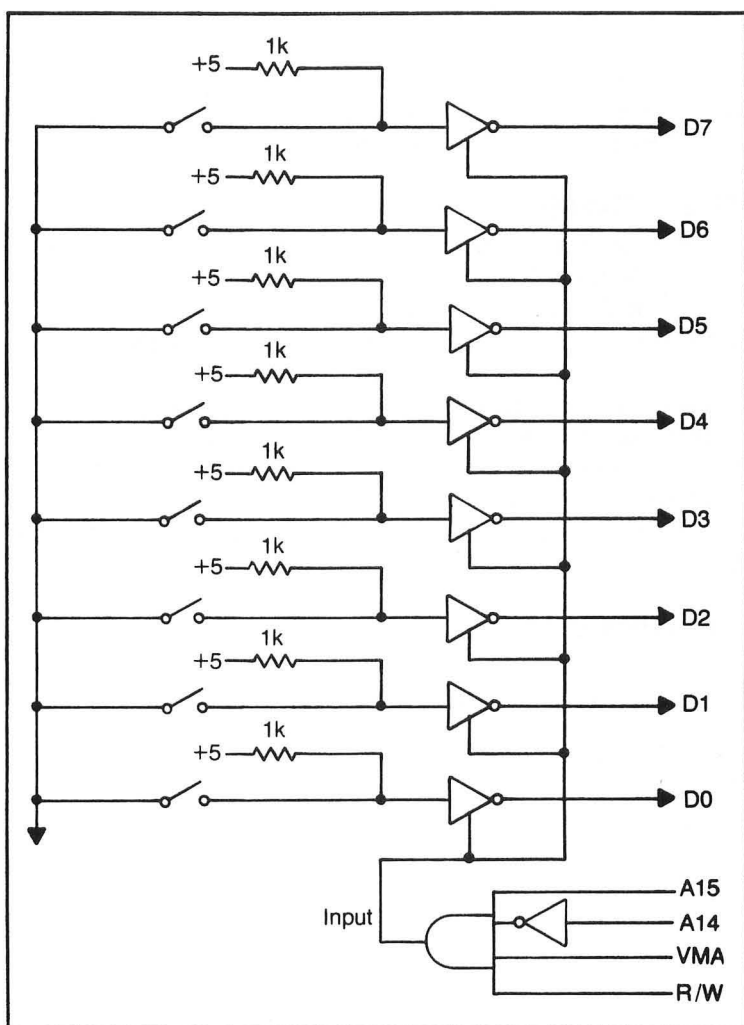


Fig. 2-5. A simple input port places the input data on the data bus only when addressed for reading with a valid memory address.

bus cycle, a central clock is required for the microprocessor, or for a part of the microprocessor IC. In either event, an external frequency-determining component is necessary.

### Storage and the Buses

The signals on the address and control buses, and on the data bus during writing are all produced directly by the microprocessor.

Sometimes there may be digital logic external to the microprocessor itself required to create all of these signals, but these external ICs are considered part of the microprocessor complex. How these signals are used determines the complexity of the input, output, and storage elements of the microcomputer.

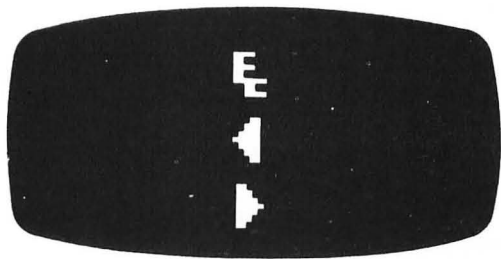
## **TYPES OF STORAGE**

Storage in a microcomputer is usually made up of read-only memory (ROM) ICs for permanent storage, and random-access memory (RAM) ICs for temporary storage. The ROMs are written once before installation in the microcomputer. Since these storage elements are not written to by the microprocessor, they can be permanently encoded with a program, and that program will be ready to execute whenever power is turned on. On the other hand, the RAM uses semiconductor ICs that can be both read and written to. However, when power is first applied their initial data contents are random. Data cannot be stored in semiconductor RAMs while the power is off.

## **BUS CONTROL OF I/O**

Input signals can be connected to the data bus for use when the instructions call for the reading of input data. A simplified version of the techniques used for storage shown in Fig. 2-5 can be used. The address bus and parts of the control bus are combined to create an input signal. When that signal is high, the input data is sent to the data bus lines.

## Chapter 3



### Computer Arithmetic

In Chapter 1 of this book you were introduced to how to use the binary number system, how to convert decimal numbers to binary numbers (and vice versa), and how to convert between other number systems. This chapter is designed to introduce you to the fundamentals of binary mathematics—addition, subtraction, multiplication, and division. Since microprocessors use binary numbers for data and control, it is extremely important that you become familiar with them.

#### ADDITION AND SUBTRACTION OF BINARY NUMBERS

Binary numbers can be added and subtracted in much the same way that the more familiar decimal numbers are. When doing binary arithmetic, remember that when you add or subtract two binary numbers, the result must be equal to the value obtained by adding or subtracting the decimal equivalents and then converting that result to a binary number.

It is obvious that we could add two binary numbers by converting the binary numbers into their decimal equivalents, adding the decimal numbers as we did in grade school, and then converting the result back into the binary numbers. However, there's an easier way. Perhaps you won't think it's easier at first, but after you become more familiar with the binary number system, you will find it so.

In the binary system, there are four rules that help to make addition quite simple. They are summarized as follows:

1.  $0 + 0 = 0$
2.  $0 + 1 = 1$
3.  $1 + 1 = 0$  with a carry of 1 equals 10
4.  $1 + 1 + 1 = 1$  with a carry of 1 equals 11

Before continuing, let's review the decimal/binary equivalents shown in Table 3-1.

With the above table in mind, let's look at the following addition problem.

$$\begin{array}{r} 1010 \\ +101 \\ \hline 1111 \end{array}$$

Looking at the conversion table, we see that binary number  $1111_2$  is equal to decimal number 15. Now let's add these numbers another way. Again, looking at the table, we note that the top number, 1010, is equal to the decimal number 10; the lower number, 101, is equal to the decimal number 5. So, the problem states  $10 + 5 = 15$ . Now the total should be converted to a binary number. The table tells us that decimal 15 is equal to binary 1111—our original answer. So we know our addition was correct.

A more complicated example of binary addition, with several carries, is shown as follows:

$$\begin{array}{r} 11111 \\ +1111 \\ \hline 101110 \end{array}$$

At first glance you might have thought the sum of the two numbers in the example above should have been 12222, but you must remember that we are working with the binary number system, and our only digits are 0 and 1. We must go back to the four rules given earlier in this chapter. Starting with the first column at the right, we add binary 1 to binary 1 (Rule #3). The result is a 0 which is written under the first column, and a 1 that must be carried to the second column. In the second column, 1 plus 1 equals 0 with a carry of 1. To this sum, you must add the 1 you carried from the first column. Thus, 0 plus 1 (Rule #2) equals 1; this result is written under the

**Table 3-1. Decimal-Binary Equivalents.**

<u>DECIMAL</u>	<u>BINARY</u>
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000

second column sum. In column 3, 1 plus 1 equals 0 with a carry of 1. To this sum, the second column carry is added. This yields a third column sum of 1 (0 plus 1 = 1). Rules 2 and 3 are used to obtain the sum in this column. In column 4, 1 plus 1 equals 0 with a carry of 1. To this sum, the third column carry is added, yielding a fourth column sum of 1 (0 plus 1 = 1). Now we have only the carry to add to the number in column 5, which gives us 10 as the sum in the fifth column. This figure was obtained by using rule #3 (1 + 1 = 0 with a carry of 1). The carry is written immediately because there are no more columns to carry it to.

If we write the equivalent decimal numbers, we'll find that 11111 in binary is equal to decimal 31, and 1111 in binary is equal to decimal 15. The sum of these two decimal numbers is 46. Decimal 46 is equal to binary 101110, proving that our binary addition is correct.

Binary addition is not limited to two figures. On the contrary, any number of figures may be added as long as the basic rules are followed. Here they are one more time.

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$1 + 1 = 10 \text{ (zero with carry of 1)}$$

$$1 + 1 + 1 = 11 \text{ (one with carry of 1)}$$

Now let's try adding three binary figures just to see how this is accomplished. Look at the following figures.

$$\begin{array}{r}
 110101 \\
 11011 \\
 + \quad 101111 \\
 \hline
 1111111
 \end{array}$$

To add these three binary numbers, start at the extreme right as before and add the first column. We have  $1 + 1$  which equals zero (with a carry of one) plus 1 which equals 1 ( $1 + 1 = 0$ ;  $0 + 1 = 1$  with a carry of 1 to the second column). Adding this carried 1 to 0 (the top digit in the second column), gives us 1 (Rule #2). This 1 added to the remaining two 1s in the column gives us 1 with a carry of 1 (Rule #4). In the third column, the carried 1 plus the top 1 in the column equals 0 with a carry of 1; 0 plus 0 (the second digit from the top) equals 0; then 0 plus 1 equals 1 (Rule #2). Thus the digit 1 is written for the sum of the third column and we have a carry of 1.

This procedure is continued until we have a total sum of 1111111 in binary. To check this addition, convert the binary numbers to decimal numbers. They are, from top to bottom, 53, 27, and 47 respectively. These three decimal numbers added together gives a sum of 127. The decimal number 127 is equal to the binary number 1111111. So our addition is correct.

In performing binary addition, do not use a pocket calculator as this will only confuse you. Also, you might find it helpful to make a small pencil mark at the top of the column to indicate each 1 carried. The carries of our last example could be indicated as follows:

$$\begin{array}{r}
 11111 \\
 110101 \\
 11011 \\
 + \quad 101111 \\
 \hline
 1111111
 \end{array}$$

You may not need these marks in simple binary addition, but in the more complex problems you'll certainly find a need for them. For example, let's look at the following addition of four binary numbers.

$$\begin{array}{r}
 11001 \\
 10101 \\
 11101 \\
 + \quad 10101 \\
 \hline
 \end{array}$$

The addition is done, and the carries are marked as follows:

$$\begin{array}{r}
 1 \ 111 \ 11 \ 11 \ 1 \ 11 \\
 \phantom{1} \phantom{111} \phantom{11} \phantom{11} \phantom{1} \phantom{11} \\
 \phantom{1} \phantom{111} \phantom{11} \phantom{11} \phantom{1} \phantom{11} \\
 \phantom{1} \phantom{111} \phantom{11} \phantom{11} \phantom{1} \phantom{11} \\
 \phantom{1} \phantom{111} \phantom{11} \phantom{11} \phantom{1} \phantom{11} \\
 \phantom{1} \phantom{111} \phantom{11} \phantom{11} \phantom{1} \phantom{11} \\
 + \phantom{1} \phantom{111} \phantom{11} \phantom{11} \phantom{1} \phantom{11} \\
 \hline
 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0
 \end{array}$$

To add these figures, first deal with the column on the extreme right as discussed previously; we have 1 plus 1 equals 0 with a carry of 1. A small 1 is then placed above the second column before we proceed to add the rest to the first column. We have another 1 plus 1 which again gives us 0 plus another 1 to carry. So a second small 1 is placed above the second column as shown in the example above. The total of the first column is therefore 0.

In the second column, we have four zeros which would total 0; then the two 1's (from our carries) are added. Since 1 plus 1 equals 0 with a 1 to carry, 0 is written as the sum of the second column, and a small 1 is placed at the top of the third column.

When the third column is added in the same manner, the sum is 0 with two 1's to carry. The sum of the fourth column is also 0 with a carry of two 1's. When we add the fifth column, we have 1 + 1, which equals 0 and with 1 to carry, so we place a 1 over the sixth column's place. We add the next 1 plus 1 which gives another 1 to be placed in the sixth column. Now we have to add the two carries, which gives us a 0 under the fifth column and a third 1 to carry to the sixth column.

In the sixth column, there are no numbers to be added—just the numbers that have been carried. One plus 1 equals 0 with a 1 to be carried to the seventh column; we place a small 1 where the seventh column would be. Now we have a 0 plus the third 1 that we carried to add. That gives us the sum of 1 for the sixth column.

The seventh column has only a single carry so it is written down at the bottom as the seventh figure in the total sum. This gives us a binary sum of 1100000.

The decimal equivalents of the binary numbers just discussed are as follows:

$$\begin{array}{r}
 25 \\
 21 \\
 29 \\
 + \ 21 \\
 \hline
 96
 \end{array}$$

## Subtraction

Binary subtraction is performed exactly as decimal subtraction is performed, so a review of decimal subtraction is in order. You know, for example, that if decimal 1827 is subtracted from 6845, the difference obtained is 5018.

$$\begin{array}{r} 6845 \text{ Minuend} \\ - 1827 \text{ Subtrahend} \\ \hline 5018 \text{ Difference} \end{array}$$

In the right-hand column, the digit 7 is larger than the digit 5 in the minuend, so a 1 is borrowed from the next higher-order digit in the minuend (the 4 in our case). The 5 in the minuend becomes 15 and the 4 becomes 3. No other borrowing is necessary in our example.

When subtracting binary numbers there are a few basic rules to remember.

1.  $0 - 0 = 0$
2.  $1 - 1 = 0$
3.  $1 - 0 = 1$

These are three of the four subtractions you will have to perform in binary. The other is to subtract 1 from 0. In this case, you must borrow from the next digit to the left as you would in decimal subtraction. Thus the fourth rule is:

$$10 - 1 = 1$$

With these basic rules in mind, let's work a few examples.

$$\begin{array}{r} 11011 \\ - 1001 \\ \hline 10010 \end{array}$$

Starting from the right-hand column,  $1 - 1 = 0$  (Rule #2) in the first column;  $1 - 0 = 1$  (Rule #3) in the second column;  $0 - 0 = 0$  (Rule #1) in the third column;  $1 - 1 = 0$  (Rule #2) in the fourth column; and in the last column,  $1 - 0 = 1$  as stated in Rule #3.

The subtraction may be checked by doing the same problem using decimal numbers. Binary 11011 is equal to 27 in decimal and binary 1001 is equal to decimal 9. Therefore,  $27 - 9 = 18$ . When you convert decimal 18 to binary, you'll find that the number is 10010, the same as our original answer. Our subtraction was correct!

You may also check binary subtraction by adding; using binary addition. In our previous example, we found the difference to be



10010 in binary. If we add this to 1001, we get an answer of 11011, again proving our subtraction correct.

Now let's look at another example which requires borrowing.

$$\begin{array}{r} 101101 \\ - 11011 \\ \hline 10010 \end{array}$$

The first column is regular subtraction, but in the second column it becomes necessary to borrow 1 from the third column. The difference for the second column will then be 1 since  $10 - 1 = 1$ . Since the 1 in the third column was transferred to the second column during the borrowing, the third column subtraction is  $0 - 0 = 0$ . One from one in the fourth column is again 0 (Rule #2). The fifth column again requires borrowing. The 1 is borrowed from the sixth column, making the problem  $10 - 1 = 1$  and completing our subtraction.

The subtraction in the preceding problem was really not too difficult since all borrowing involved adjacent columns. In the following problem, however, we must borrow from the third column, making the value in the second column 10. From the binary 10 we have in the second column, we borrow 1, giving us 10 in the first column and leaving 1 in the second column. The rest of the problem now becomes simple subtraction.

$$\begin{array}{r} 11100 \\ - 1001 \\ \hline 10011 \end{array}$$

## MULTIPLICATION AND DIVISION OF BINARY NUMBERS

Multiplication is a short method of adding a number to itself the number of times specified by the multiplier. Binary multiplication follows the same general principles as decimal multiplication. However, with only two possible multipliers (1 or 0), binary multiplication is much simpler than decimal multiplication. As with binary addition and subtraction, there are a few simple rules to follow when multiplying binary numbers.

1.  $0 \times 0 = 0$
2.  $0 \times 1 = 0$
3.  $1 \times 0 = 0$
4.  $1 \times 1 = 1$

As in addition and subtraction, once the multiplication is performed in binary, the answer may be checked by converting all numbers to their decimal equivalents and doing the problem again.

Let's multiply the binary number 1010 by binary 11; the problem is written as follows:

$$\begin{array}{r}
 1010 \text{ Multiplicand} \\
 \times 11 \text{ Multiplier} \\
 \hline
 1010 \\
 1010 \\
 \hline
 11110 \text{ Product}
 \end{array}$$

To multiply, we use the right-hand digit in the multiplier to multiply each digit in the multiplicand. So, using the four basic rules as a guide,  $1 \times 0 = 0$ ,  $1 \times 1 = 1$ ,  $1 \times 0 = 0$ , and  $1 \times 1 = 1$  or 1010. The second 1 in the multiplier is used to multiply each digit in the multiplicand again, just as in decimal multiplication. All that's left to do now is to add using the rules we learned earlier. Our answer is 11110.

Binary multiplication becomes somewhat more difficult when we have to do a series of carries. For example, let's multiply the following binary numbers:

$$\begin{array}{r}
 1110 \\
 \times 111 \\
 \hline
 \end{array}$$

In this example, the multiplication is standard, but the addition of the three subproducts is complicated by carries. To illustrate,

$$\begin{array}{r}
 1110 \\
 \times 111 \\
 \hline
 1110 \\
 1110 \\
 1110 \\
 \hline
 1100010
 \end{array}$$

Using the four rules of binary addition, the first 0 is merely brought down. In the second column,  $1 + 0 = 1$ . In the third column,  $1 + 1 = 0$  with a carry of 1. In the fourth column,  $1 + 1 = 0$  with a carry of 1; then 1 plus the 1 carried from the third column = 0 with another carry of 1. In the fifth column,  $1 + 1 = 0$  with a carry of 1. Then, adding the two 1's carried from the fourth column, we get 0 with

another carry of one. In the sixth column,  $1 + 1 + 1$  equals 1 with a carry of 1 (or 11 since there are no other digits to add in the seventh column). The answer, therefore, is 1100010.

If the process seems difficult to you at first, practice several problems with carries, and then check your answer by converting the problems to the decimal number system. If any problem does not work out correctly, be sure to repeat it until you locate your mistake.

## Binary Division

Division is the reverse of multiplication. Therefore, since multiplication primarily involves addition, division primarily involves subtraction. If you learned the basic principles of binary subtraction earlier in this chapter, you should have little trouble doing binary division.

The numbers in binary division are set up in much the same way as decimal numbers are set up in conventional decimal division. For example, in dividing binary 100 into binary 1000, the problem is set up as follows:

$$100 \overline{)1000}$$

In the above example, the divisor is 100 and the first three digits in the dividend are 100, so 100 will go into these three digits once. Then the divisor is multiplied by this 1 and the results written beneath the first three digits. Now we subtract. The remaining 0 is brought down. The divisor (100) will go into 0 zero times, so we place a 0 in the quotient for a total of 10.

$$\begin{array}{r} 10 \\ 100 \overline{)1000} \\ \underline{100} \phantom{00} \\ 00 \end{array}$$

Now look at the following example. Here 101 is divided into 1111.

$$\begin{array}{r} 11 \\ 101 \overline{)1111} \\ \underline{101} \phantom{00} \\ 101 \phantom{00} \\ \underline{101} \phantom{00} \\ 0 \end{array}$$

Binary 101 goes into binary 111 once, so we place a 1 above the third 1 from the left. Then we subtract 101 from 111 leaving 10. The remaining 1 in the dividend is brought down and the divisor (101) goes into this 1 time with no remainder. Thus, the result of this division is binary 11.

The preceding examples were relatively short and quite simple, but they should suffice to introduce you to binary division. In longer problems, the division is essentially the same. There may be many more steps and more chances to make errors, but the division is done the same way. It is important to do your borrowing when you subtract. When you borrow 1 from 1, for example, you get binary 10 and leave 0 behind. If you borrow 1 from 10, you get 10, and leave 1 behind. To remind yourself that you have borrowed, put the appropriate figure near the original digit. If you don't do this, chances are that you will forget what the digit should be and make a mistake.

Learning binary arithmetic is mostly a matter of getting practice, so do several problems daily until the procedure becomes second nature to you. Always convert the binary numbers to decimal ones and check your math for errors. If any are found, rework the problem until you find your mistakes.

## FRACTIONS AND DECIMALS

Converting decimal fractions and decimals to their binary equivalents is not quite as easy as converting whole numbers. The value of the negative powers of 2 decreases as the exponent increases, whereas the positive powers of 2 are just the opposite. Nevertheless, fractions and decimals can be converted to their binary equivalents.

In some cases, just like within the decimal system, an exact equivalent cannot be obtained. Take, for example the decimal fraction  $\frac{1}{3}$ ; it equals .333... and no matter how far we carry this expression, it will never work out evenly. While it's true that the more 3's that are added to the right of the decimal point, the more accurate the decimal expression of the fraction becomes, we will never get an exact equivalent of the fraction  $\frac{1}{3}$ . The inability to convert some fractions to an exact "decimal" equivalent should be no problem in either the binary or decimal number systems. The decimal equivalent can simply contain a sufficient number of digits to give the particular degree of accuracy needed.

A complete table showing the negative powers of 2 up to  $2^{-10}$  is given in Table 3-2. Note that as the negative exponent increases, the value of the fraction and the decimal equivalent decreases.

**Table 3-2. Negative Powers of 2.**

$2^{-1}$	$=$	$\frac{1}{2}$	$=$	0.5
$2^{-2}$	$=$	$\frac{1}{4}$	$=$	0.25
$2^{-3}$	$=$	$\frac{1}{8}$	$=$	0.125
$2^{-4}$	$=$	$\frac{1}{16}$	$=$	0.0625
$2^{-5}$	$=$	$\frac{1}{32}$	$=$	0.03125
$2^{-6}$	$=$	$\frac{1}{64}$	$=$	0.015625
$2^{-7}$	$=$	$\frac{1}{128}$	$=$	0.0078125
$2^{-8}$	$=$	$\frac{1}{256}$	$=$	0.00390625
$2^{-9}$	$=$	$\frac{1}{512}$	$=$	0.001953125
$2^{-10}$	$=$	$\frac{1}{1024}$	$=$	0.0009765625

Table 3-3 gives the binary equivalents of a set of fractions (sixteenths). You can see that we work with fractional binary numbers in essentially the same way as we do with whole numbers. As we work from the binary point to the left in whole binary numbers, the value increases. As we go from the binary point to the right in fractional binary numbers, the value of each term decreases.

The addition and subtraction of binary fractions is done in the same way as the addition and subtraction of whole binary numbers. Carrying and borrowing procedures are the same, even across the binary point.

When adding mixed binary numbers and whole binary numbers, be sure to line up the binary points and to add the columns one at a time.

The problems of multiplying and dividing binary fractional numbers are essentially the same as those encountered when working with our regular decimal numbering system. Division problems are handled the same way as decimal divisions: when

$\frac{1}{16}$	= 0.0625	= 0.001
$\frac{1}{8}$	= 0.125	= 0.0010
$\frac{3}{16}$	= 0.1875	= 0.0011
$\frac{1}{4}$	= 0.250	= 0.0100
$\frac{5}{16}$	= 0.3125	= 0.0101
$\frac{3}{8}$	= 0.375	= 0.0110
$\frac{7}{16}$	= 0.4375	= 0.0111
$\frac{1}{2}$	= 0.500	= 0.1000
$\frac{9}{16}$	= 0.5625	= 0.1001
$\frac{5}{8}$	= 0.625	= 0.1010
$\frac{11}{16}$	= 0.6875	= 0.1011
$\frac{3}{4}$	= 0.750	= 0.1100
$\frac{13}{16}$	= 0.8125	= 0.1101
$\frac{7}{8}$	= 0.875	= 0.1110
$\frac{15}{16}$	= 0.9375	= 0.1111
$\frac{16}{16}$	= 1.000	= 1.0000

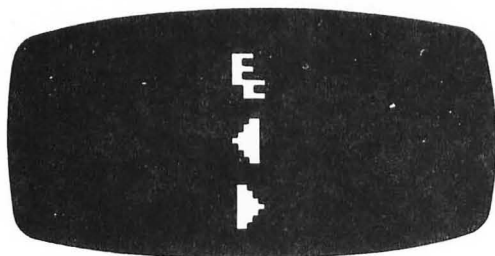
**Table 3-3. Binary  
Equivalents of Fractions in Sixteenths.**

dividing with a decimal number, get rid of the decimal in the divisor by moving the decimal point to the right in both the divisor and the dividend. In other words, you have the same basic problem in both number systems—keeping track of the decimal point. Here are some examples of binary multiplication and division.

$$\begin{array}{r}
 10.1 \\
 \times \quad \frac{10.1}{10.1} \\
 \hline
 101 \\
 \hline
 110.01
 \end{array}$$

$$\begin{array}{r}
 0.01 \overline{)0.0001} \\
 \phantom{0.01} \frac{0.01}{1} \overline{)0.01} \\
 \phantom{0.01} \phantom{0.01} \frac{1}{0}
 \end{array}$$

## Chapter 4



### Boolean Operations

Along with the basic mathematical processes examined earlier, the microprocessor in the ATARI 800 can manipulate binary numbers logically. This system was conceived using the theorems developed by the English mathematician George Boole (1815-1864). Boolean values are used in the process of reasoning (Boolean algebra), or in a deductive system of theorems using a symbolic logic, and dealing with classes, propositions, or on-off circuit elements. Boolean algebra employs symbols to represent operations such as AND, OR, NOT, EXCEPT, and in some cases (in programming) if ...then to permit mathematical calculation.

The simplest kind of Boolean expression states a relationship which may or may not be true, between two quantities. If the quantities are numeric, then the possible relations are strict inequalities;  $<$ ,  $>$ ; and reflexive inequalities: less than or equal,  $<=$ , and greater than or equal,  $>=$ , equals,  $=$ ; and does not equal,  $\neq$ . If the quantities are non-numeric, for instance, the address of apartments, it makes no sense to talk about inequalities which derive from an ordering of values, but the relationships "equals" and "does not equal" are still meaningful.

A relational expression results in a Boolean value, 1 or 0, depending on whether the two quantities compared in the relational expression do or do not satisfy the given relation. If variables or numerical expressions are written to represent the related quantities, the values of these variables or expressions are first obtained



and then examined to determine whether or not the indicated relationship holds for these values.

Relational expressions are the most obvious, but they are not the only kind of Boolean expression. We sometimes wish to base a decision on whether or not several relational conditions hold simultaneously or on whether or not at least one of several possible conditions hold. Consider the following:

The U.S. Postal service has recently imposed restrictions on the maximum size and weight of a package to be sent by parcel post. The weight must not exceed 44 pounds, and the sum of the length plus the girth must not exceed 72 inches. Assume that for parcels of rectangular cross-section, the dimensions are given as values of the variables A, B, and C in units of inches, and the weight as the value of WGHT in pounds. However, the dimensions are not necessarily given in order from greatest to least. We must write a procedure to determine whether or not any given parcel is acceptable.

In order to calculate the postal dimension, we need to know which of the three dimensions is the largest, for the equation restricts girth more than it does length. We can compute a value for the postal dimension by making use of conditional assignment.

```
IF A = B & A = C THEN
  POSTALDIMENSIONS = A + 2 * (B+C);
ELSE IF B = C THEN
  POSTALDIMENSION = B + 2 * (A+C)
ELSE
  POSTALDIMENSION = C + 2 * (A+B);
```

We won't take time to investigate the entire program; rather, let's look at the first line of the statement. Here two conditions are tested to find out whether A is the largest of the three dimensions. If it is, then the expression  $A + 2 * (B + C)$  is evaluated and assigned to the variable. Otherwise, if one or the other of the conditions tested in the first line is false, the assignment on the second line is not evaluated, but instead control passes beyond the first ELSE. The second alternative is tried only after it has been determined that A cannot be the largest of the three dimensions, etc.

## **BOOLEAN ALGEBRA**

Before going further into conditional execution, the reader should have a good knowledge of Boolean Algebra. This will help him or her to approach the work more intelligently.

In the spring of 1847, George Boole wrote a pamphlet entitled "A Mathematical Analysis of Logic." Later, in 1854, he wrote a more exhaustive treatise on this subject which was entitled. "An Investigation of the Laws of Thought." It is this later work which forms the basis for our present day mathematical theories used for the analysis of logical processes.

Although conceived in the 18th Century, little practical application was found for Boolean algebra until 1938 when it was found that Boolean algebra could be adapted for the analysis of telephone relay and switching circuits. Since that time, the extent of its use has expanded rapidly, roughly paralleling the development and use of more complex switching circuits such as those found in present day automatic telephone dialing systems and digital computers. Thus Boolean algebra has become an important subject which must be learned if the operation of digital computers and other devices using complex switching circuits is to be understood.

## **CLASSES AND ELEMENTS**

In our universe, it is logical to visualize two divisions; all things of interest in a discussion are in one division and all things which are not of interest in the discussion are in the other division. These two divisions comprise a set or class which is designated the universal class, and all things contained in the universal class are called elements. One may also visualize another set or class; this class contains no elements and has been designated the null class.

In a particular discussion, certain elements of the universal class may be grouped together to form combinations which are known as classes. However, these classes are actually subclasses of the universal class and thus should not be confused with the universal class or the null class. Each subclass of the universal class is dependent on its elements and the possible states (stable, unstable, or both) that these elements may take.

Boolean algebra is limited to the use of elements which have only two possible states, both of which are stable. These states are usually designated TRUE (1) or FALSE (0). Thus, to determine the number of classes (or possible combinations of elements) in Boolean Algebra, we find the numerical value of  $2^n$  when  $n$  equals the number of elements. If we have two elements (each element has two possible states), we have  $2^2$  possible classes. If the elements are designated A and B, then A may be true or false and B may be true or false. Using the connective word "and," the classes which could be formed are as follows:

- A true and B false
- A true and B true
- A false and B true
- A false and B false

However, if the connective word “or” is used, four additional classes are formed. The differences between the two classes, groups or forms are discussed below.

## VENN DIAGRAMS

The Venn diagram is a topographical picture of logic, composed of the universal class divided into classes depending on the number of elements. Venn diagrams may be used to illustrate Boolean logic as follows:

Consider the universal class as containing submarines and atomic-powered items. Let A equal submarines and B equal atomic-powered items. We have four classes which are:

- (1) Submarines and not atomic powered
- (2) Submarines and atomic powered
- (3) Atomic powered and not submarines
- (4) Not submarines and not atomic powered

These four classes are called minterms since they represent the four minimum classes of elements. The opposite of the minterms are the maxterms which are stated as follows:

- (1) Atomic powered or not submarines
- (2) Not submarines or not atomic powered
- (3) Submarines or not atomic powered
- (4) Submarines or atomic powered

The various relationships which may exist are represented by the Venn diagrams in Fig. 4-1.

## CONNECTIVES AND VARIABLES

Before proceeding it will be necessary to identify and define the symbology used in Boolean algebra. As may be seen, most of these symbols are common to other branches of mathematics; however in Boolean algebra they may have a slightly different meaning or application.

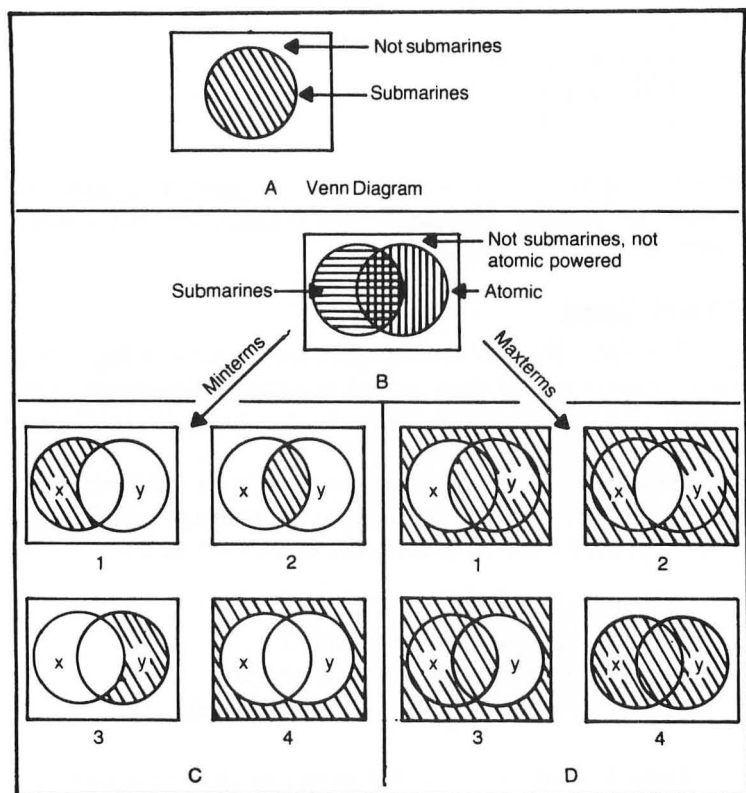


Fig. 4-1. The Venn diagram.

=

The equal sign, just as in conventional mathematics, represents a relationship of equivalence between the expressions so connected.

or x

The dot or small x indicates the logical product, or conjunction of the terms so connected. The operation is also frequently indicated with no symbol use, i.e.,  $A \cdot B = A \times B = AB$ . Most generally referred to as the AND operation, the terms so related are said to be "ANDed."

+

The plus sign indicates the logical sum operation, a disjunction of the terms so connected. Usually called the OR operation and the terms so connected are said to be "ORed."

---

	<p>The <i>vinculum</i> serves a dual purpose. It is at the same time a symbol of grouping and of operation. As a sign of operation it indicates that the term(s) so overlined are to be <i>complemented</i>. Complement can be defined as "a Boolean operation whose result is the same as that of another operation but with the opposite sign; thus, OR and NOR operations are complementary." As a symbol of grouping it collects all terms to be complemented together. Terms so overlined are often said to be negated, the process of taking the complement is then called negation.</p>
( )	<p>These familiar signs of grouping are used in the customary fashion to indicate that all terms so contained are to be treated as a unit.</p>
A,B, etc.	<p>Various letters are used to represent the variables under consideration, generally starting with A. Since the variables are capable of being in only one of two states the numerals 0 and 1 are the only numbers used in a Boolean expression.</p>

## APPLICATIONS TO SWITCHING CIRCUITS

Since Boolean algebra is based upon elements having two possible stable states, it becomes very useful for analyzing switching circuits. The reason for this is that a switching circuit can be in only one of two possible states. That is, it is either open or it is closed. We may represent these two states as 0 and 1 respectively. The basic switching operations are discussed below. All other switching operations (even the most complex) are merely combinations of these basic operations.

Let us consider the Venn diagram in Fig. 4-2A. Its classes are labeled using the basic expressions of Boolean algebra. Note that there are two elements, or variables, A and B. The shaded area represents the class of elements that are  $A \cdot B$  in Boolean notation and is expressed as:

$$f(A,B) = A \cdot B$$

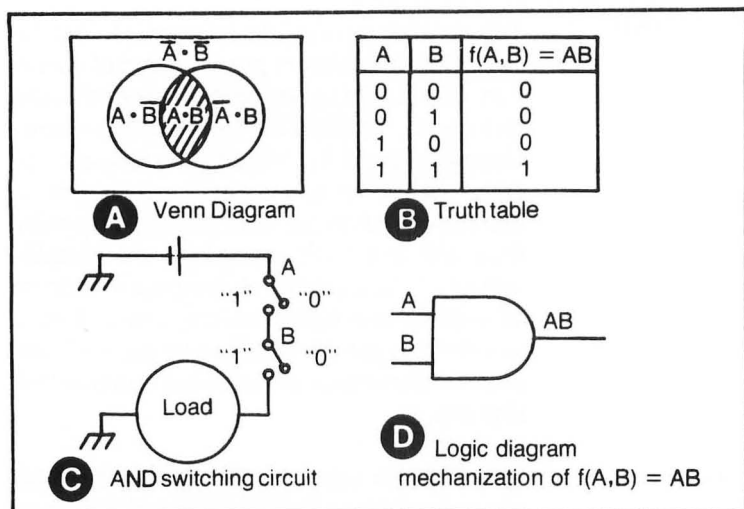


Fig. 4-2. The AND operation.

The other three classes are also indicated in Fig. 4-2A. This expression is called an AND operation because it represents one of the four minterms previously discussed. Recall that AND indicates class intersection and both A and B must be considered simultaneously.

We can conclude then that a minterm of  $n$  variables is a logical product of these  $n$  variables with each variable present in either its noncomplemented or its complemented form. It is considered an AND operation.

For any Boolean function there is a corresponding truth table which shows, in tabular form, the true conditions of the function for each state its variables can be in. In Boolean algebra, 0 and 1 are the symbols assigned to the variables of any function. Figure 4-2B shows the AND operation function of two variables and its corresponding truth table.

You can see that this function is true if you think of the logic involved;  $AB$  is equal to A and B which is the function  $f(A,B)$ . Thus, if either A or B takes the condition of 0, or both take this condition, the function  $f(A,B) = AB$  is equal to 0. But if both A and B take the condition of 1, the AND operation function has the condition of 1.

Figure 4-2C shows a switching circuit for the function  $f(A,B) = AB$ . There will be an output only if both A and B are closed. An output in this case equals 1. If either switch is open (0 condition), there will be no output or 0.

In any digital equipment, there will be many circuits like the one shown in Fig. 4-2C. In order to analyze circuit operation, it is necessary to refer frequently to these circuits without looking at their switch arrangements. This is done by doing a logic diagram mechanization as shown in Fig. 4-2D. This figure indicates that there are two inputs, A and B, into an AND operation circuit producing the function (in Boolean algebra form) of  $AB$ . This diagram simplifies the circuit diagram by indicating operations without drawing all the circuit details.

It should be understood that while the previous discussion concerning the AND operation dealt with only two variables the same ideas can be applied to any number of variables. For example, in Fig. 4-3 three variables are shown along with their Venn diagram, truth table, switching circuit, and logic diagram mechanization.

Consider the Venn diagram in Fig. 4-4A; note that there are two elements, or variables, A and B. The shaded area represents the class of elements that are  $A+B$  in Boolean notation and is expressed in Boolean algebra as:

$$f(A,B) = A + B$$

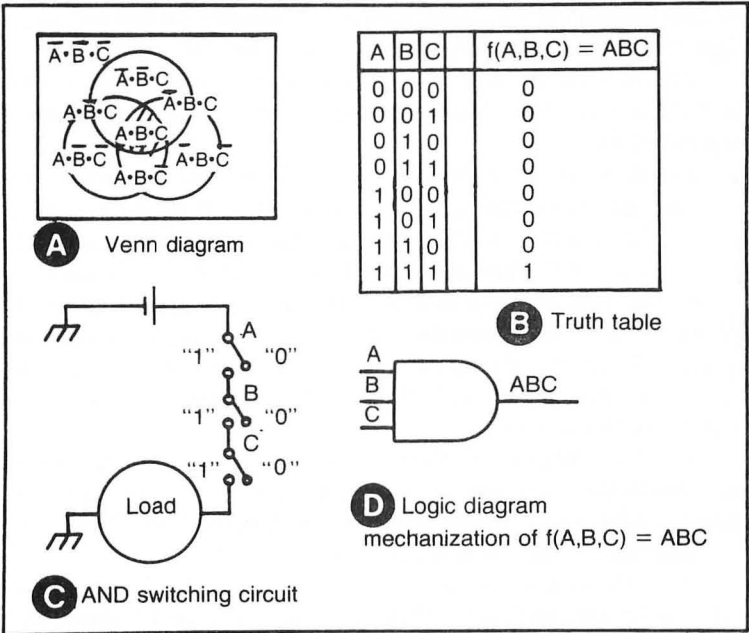


Fig. 4-3. The AND operation (three variables).

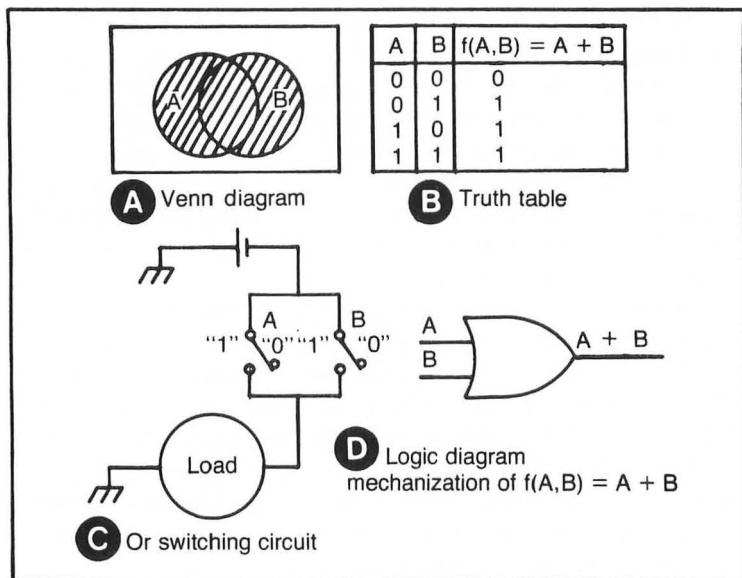


Fig. 4-4. The OR operation.

This expression is called an OR operation for it represents one of the four maxterms previously discussed. Recall that OR indicates class union, and either A or B or both must be considered. A maxterm of  $n$  variables is a logical sum of these  $n$  variables where each variable is present in either its noncomplemented or its complemented form.

In Fig. 4-4B the truth table of an OR operation is shown. You can understand this truth table if you think of  $A+B$  as being equal to A or B. Thus if either A or B takes the value 1,  $f(A,B)$  must equal 1. If neither A nor B takes the value 1, the function equals zero.

Figure 4-4C shows a switching circuit for the OR operation. It contains two or more switches in parallel. It is apparent that the circuit will transmit if either A or B is in a closed position (that is, equal to 1). If, and only if, both A and B are open (equal to 0) the circuit will not transmit.

The logic diagram for the OR operation is given in Fig. 4-4D. This means that there are two inputs, A and B, into an OR operation circuit producing the function in Boolean form of  $A+B$ . Note how this diagram is different from that in Fig. 4-2D.

As in the discussion of the AND operation, the OR operation may also be used with more than two inputs. Figure 4-5 shows the OR operation with three inputs.



The shaded area in Fig. 4-6A represents the complement of A which in Boolean algebra is  $\bar{A}$  and read as "NOT A". The expression  $f(A)$  equals  $\bar{A}$  is called a NOT operation. The truth table for the NOT operation (Fig. 4-6B) is explained by the NOT switching circuit. A NOT circuit must take a signal that is injected at the input and produce the complement of this signal at the output. Thus, in Fig. 4-6C it can be seen that when switch A is closed (equal to 1), the relay opens the circuit to the load. When switch A is open (equal to 0) the relay completes a closed circuit to the load. The logic diagram for the NOT operation is given in Fig. 4-6D. It shows that A is the input to a NOT operation circuit and gives an output of  $\bar{A}$ . The NOT operation may be applied to any operation circuit such as AND or OR.

The shaded area in Fig. 4-7A represents the quantity A OR B negated. This figure represents the minterm expression  $\overline{A+B}$ ; that is, A OR B negated is  $\overline{A+B}$  and by application of DeMorgan's Theorem is equal to  $\bar{A}\bar{B}$ .

The truth table for the NOR operation is shown in Fig. 4-7B. The table shows that if neither A or B is equal to 1, then  $f(A,B)$  is equal to 0. Furthermore, if A and B are equal 0, then  $f(A,B)$  equals 1.

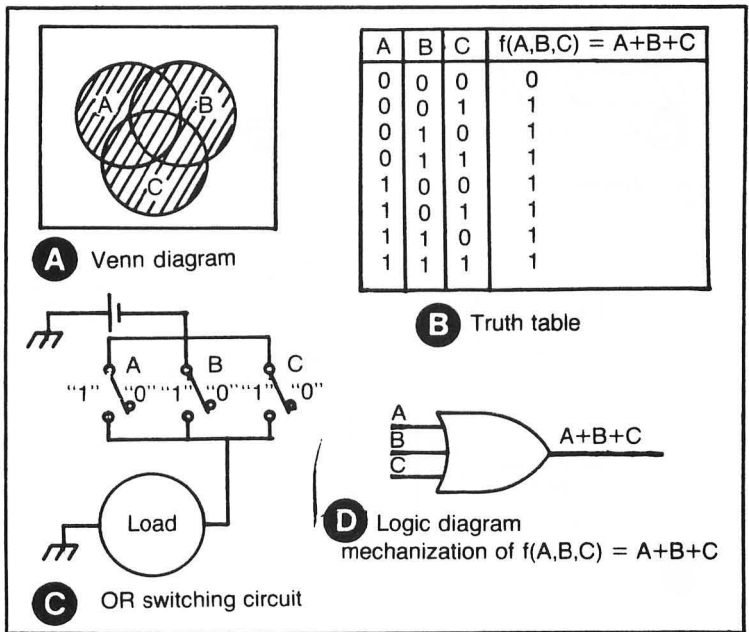


Fig. 4-5. The OR operation (three operations).

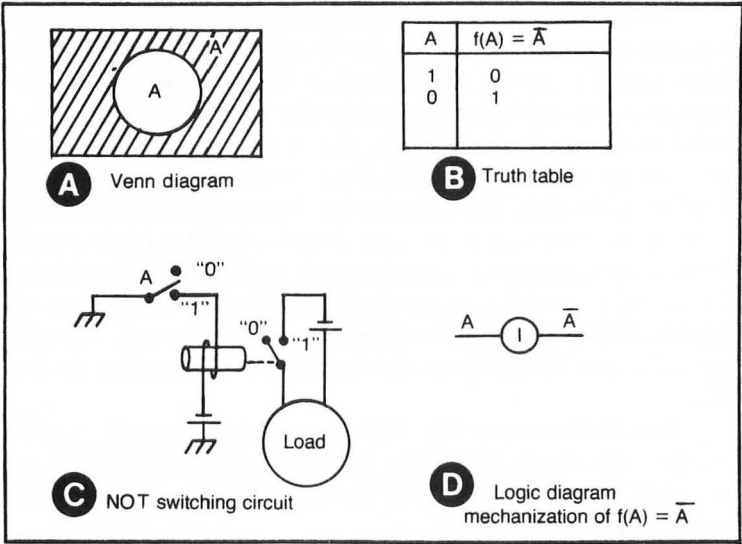


Fig. 4-6. The NOT operation.

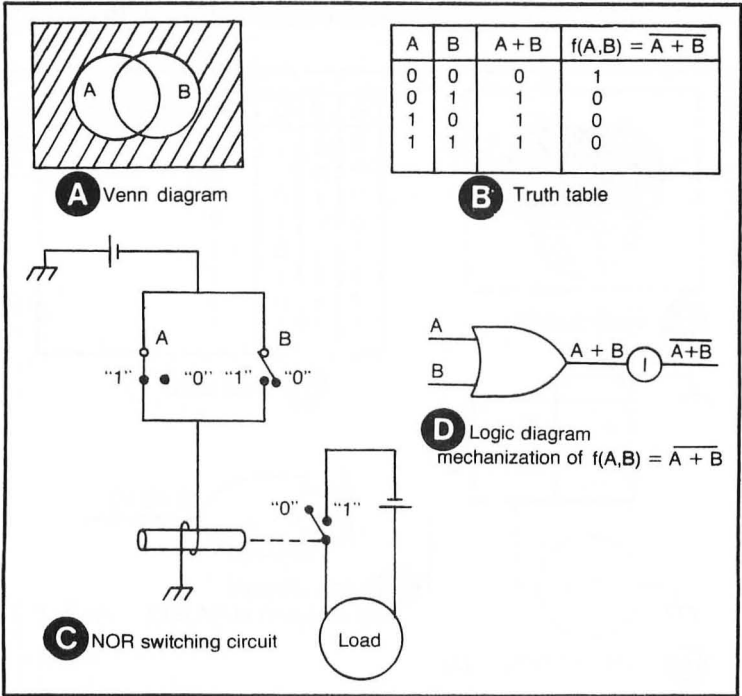


Fig. 4-7. The NOR operation.

The NOR operation is a combination of the OR operation and the NOT operation. The NOR switching circuit in Fig. 4-7C is the OR circuit placed in series with the NOT circuit. If either switch A, switch B, or both are in the closed position (equal to 1), then there is no transmission to the load. If both switches A and B are open (equal to 0), then current is transmitted to the load.

The logic diagram mechanization of  $f(A,B) = \overline{A+B}$  (NOR operation) is shown in Fig. 4-7D. It uses both the OR logic diagrams and the NOT logic diagrams. The NOR logic diagram mechanization shows there are two inputs, A and B, into an OR circuit producing the function in Boolean form of  $A+B$ . This function is the input to the NOT (inverter) which gives the output, in Boolean form, of  $\overline{A+B}$ . Note that the whole quantity of  $A+B$  is complemented and not the separated variables.

The shaded area in Fig. 4-8A represents the quantity A AND B negated (NOT), and is a maxterm expression. Notice that  $\overline{AB}$  is equal to the maxterm expression  $\overline{A+B}$ .

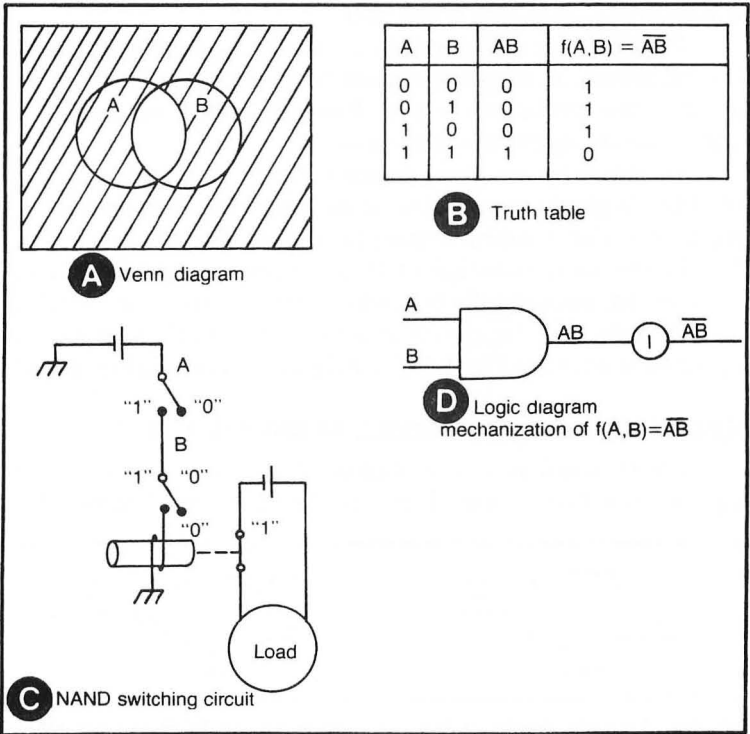


Fig. 4-8. The NAND operation.

The truth table is shown for the NAND operation in Fig. 4-8B. When A and B equal 1, then  $f(A,B)$  is equal to 0. In all other cases, the function is equal to 1.

The NAND operation is a combination of the AND operation and the NOT operation. The NAND switching circuit is shown in Fig. 4-8C. Note that the AND circuit is in series with the NOT circuit. If either switch A or B is open (equal to 0), then current is transmitted to the load. If both switches A and B are closed (equal to 1), then there is no transmission to the load.

The logic diagram mechanization of  $f(A,B) = \overline{AB}$  (NAND operation) is shown in Fig. 4-8D. The AND part of the diagram and the NOT part of the diagram show that there are two inputs, A and B, put into the AND circuit. The result is then input to the NOT circuit which gives the output, in Boolean form, of  $\overline{AB}$ . Note that the entire quantity  $AB$  is complemented and not the separate variables.

NOTE: The logic diagrams in Figs. 4-7 and 4-8 do not conform to the American Standard Logic Symbology (MIL-STD). They were drawn in this fashion to illustrate a point in the text, normally they will be drawn as shown in Fig. 4-9.

The exclusive OR operation is actually a special application of the OR operation. In this operation either A or B must be true in order for the function to be true; however, if both are true at the same time the function will be false.

As shown by the Venn diagram (Fig. 4-10A), this operation must be assigned a special class since it does not conform to any of the minterm or maxterm classes previously discussed.

In the mechanization of this operation (Fig. 4-10C) the switches are mechanically linked together so that one or the other, but not both, may be closed at a time. The truth table for this operation is shown in Fig. 4-10B and the logic symbol in Fig. 4-10D.

## FUNDAMENTAL LAWS AND AXIOMS OF BOOLEAN ALGEBRA

The laws and axioms of Boolean algebra are used to simplify any Boolean expression. They are listed below. Figures 4-11

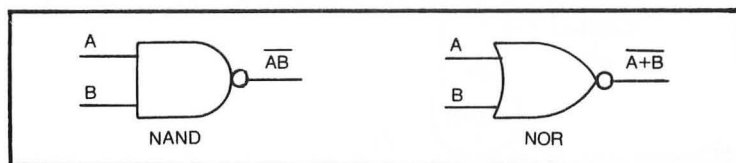


Fig. 4-9. American Standard logic symbology for the NOR and the NAND operations.

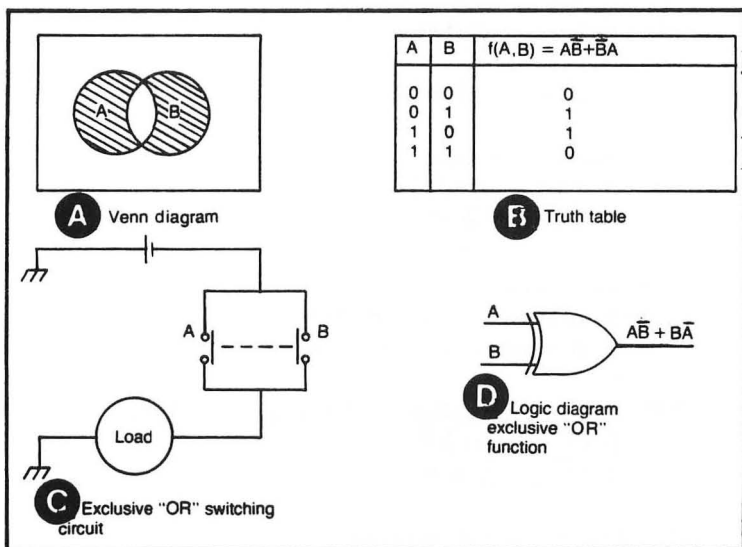


Fig. 4-10. The EXCLUSIVE OR operation.

through 4-20 show the truth tables, logic diagrams, and mechanization for these laws and axioms.

### I. Law of Identity

$$A = A$$

### II. Law of Complementarity

$$1. A\bar{A} = 0$$

$$2. A + \bar{A} = 1$$

### III. Idempotent Law

$$1. AA = A$$

$$2. A + A = A$$

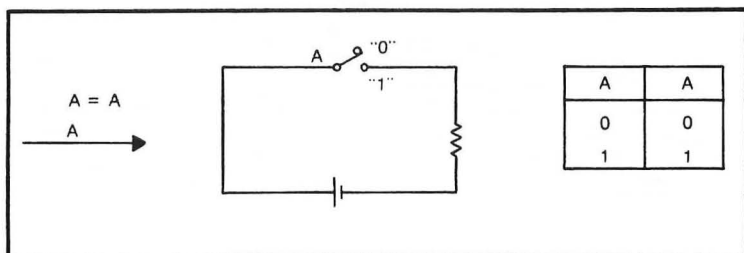


Fig. 4-11. Law of identity.

#### IV. Commutative Law

1.  $AB = BA$
2.  $A + B = B + A$

#### V. Associative Law

1.  $(AB)C = A(BC)$
2.  $(A + B) + C = A + (B + C)$

#### VI. Distributive Law

1.  $A(B+C) = (AB) + (AC)$
2.  $A + (BC) = (A + B)(A + C)$

#### VII. Law of Dualization (DeMorgan's Theorem)

1.  $\overline{(A + B)} = \bar{A}\bar{B}$
2.  $\overline{(AB)} = \bar{A} + \bar{B}$

#### VIII. Law of Double Negation

$$\overline{\bar{A}} = A$$

#### IX. Law of Absorption

1.  $A(A + B) = A$
2.  $A + (AB) = A$

#### Axioms

1.  $A + 0 = A$
2.  $A \cdot 0 = 0$
3.  $A + 1 = 1$
4.  $A \cdot 1 = A$

(The variable A may be 1 or 0.)

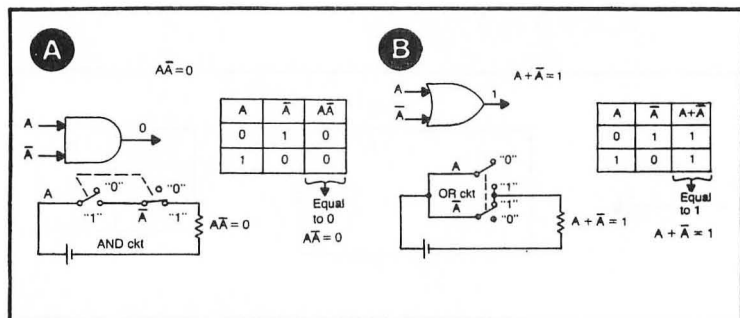


Fig. 4-12. Complementary law.

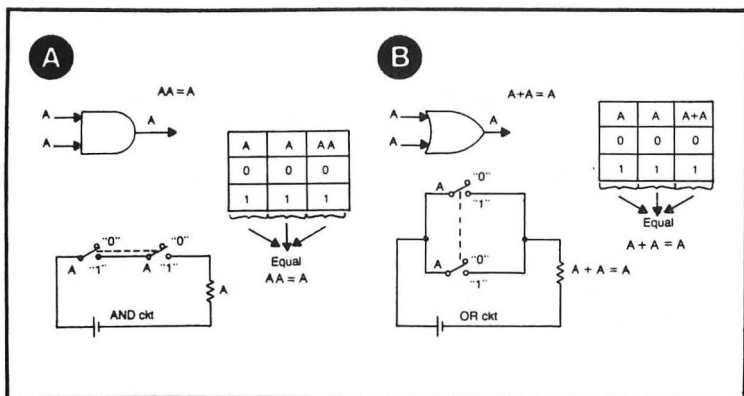


Fig. 4-13. Idempotent law.

## BOOLEAN EQUATION SIMPLIFICATION AND MECHANIZATION

As you have seen, Boolean algebra comprises a set of axioms and theorems which are useful in describing logic equations such as those used in computer technology. Likewise, these laws and axioms are used to simplify logic equations so that logic conditions can be designed in their simplest and most economic form. For example, the equation below is a logic equation which describes the logic circuit (Fig. 4-21A) in Boolean terms.

$$f = AC + AD + BC + BD$$

Boolean algebra can be used to simplify the logic equation.

$$\begin{aligned} f &= AC + AD + BC + BD \\ &= A(C + D) + B(C + D) \\ &= (A + B)(C + D) \end{aligned}$$

The logic circuit arrangement for the simplified expression is shown in Fig. 4-21B. Factors such as the loading and standardization of logic circuits may dictate the use of other than the simplest possible Boolean expression. In this discussion, the only concern is the simplification of equations. Other design considerations are not dealt with.

Consider the following as a second example:

EXAMPLE: Simplify the logic equation,

$$f = ABC + ABD + AC + ABCD + AC$$

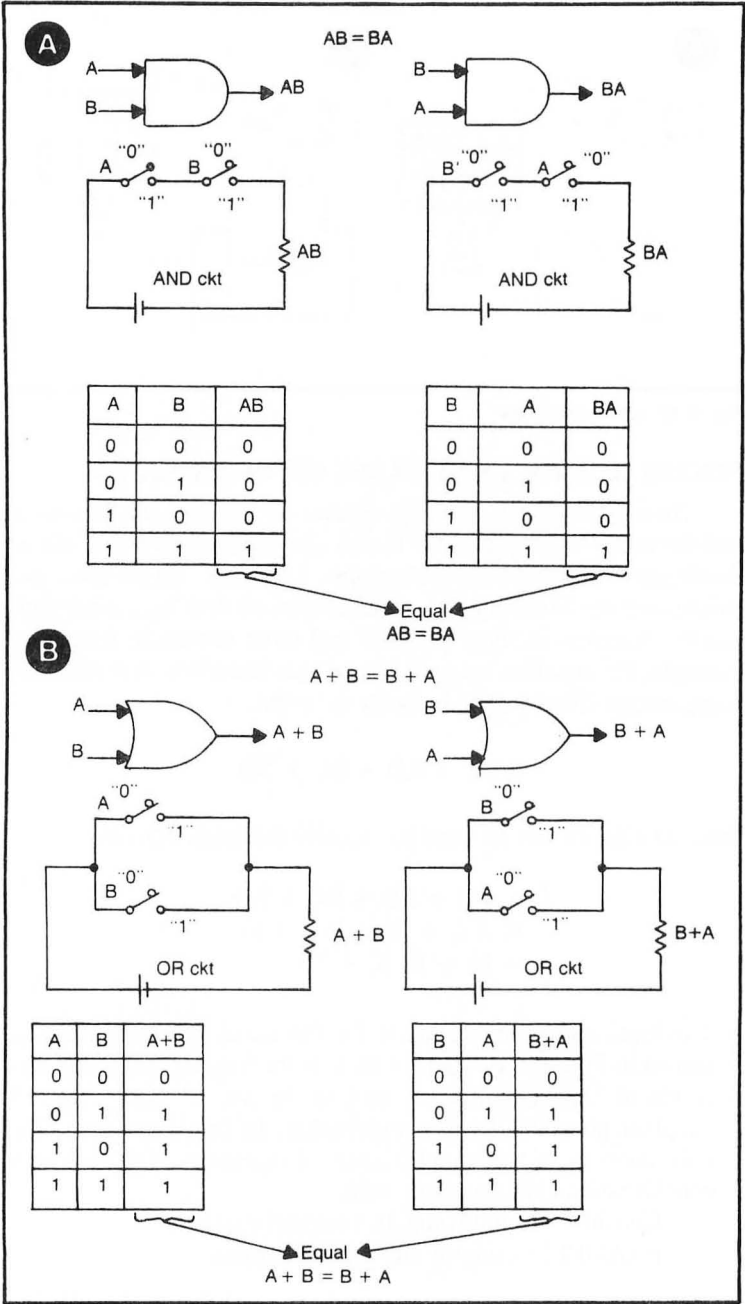


Fig. 4-14. Commutative law.



SOLUTION: Rearrange terms and factor as follows:

$$\begin{aligned}f &= ABC + AC + ABCD + AC + ABD \\&= A(BC + C) + A(BCD) + ABD\end{aligned}$$

Apply the complementary law to  $(BC + C)$  and  $(BCD + C)$ . Then:

$$f = A(B + C) + A(BD + C) + ABD$$

Apply distributive law. Then:

$$\begin{aligned}f &= AB + AC + ABD + AC + ABD \\f &= (AB + ABD) + AC + AC + ABD\end{aligned}$$

Apply the law of absorption to  $(AB + ABD)$  and rearrange terms. Then:

$$f = AB + AC + AC + ABD$$

This equation is the easiest to mechanize. However, the simplification process could be carried one step further by factoring, in which case:

$$f = A(B + C) + A(C + BD)$$

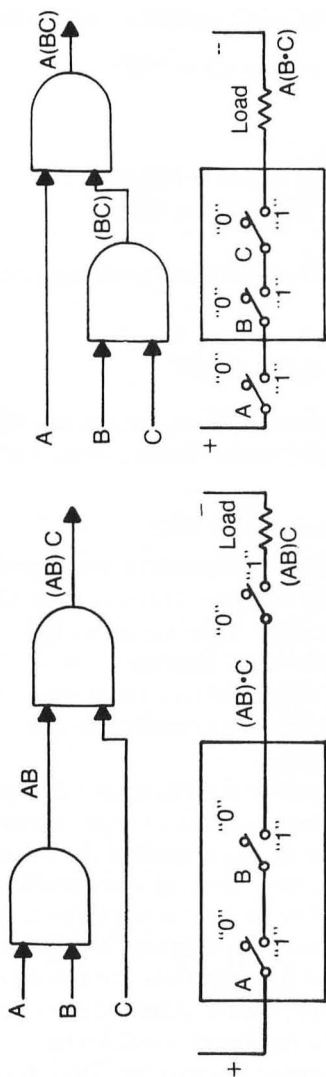
The foregoing examples of simplification show the process to be rather difficult at first. For the beginner there is no positive indication that the simplest possible logic equation has been reached. You must learn by experience. Repeated use of these theorems is the only solution. Simplification theorems are of greatest value in the preliminary stages of simplification, or in the simplification of elementary functions.

A second way approach to equation simplification is to use the Veitch diagram. This type of diagram provides a very quick and easy way for finding the simplest logic equation needed to express a given function. Veitch diagrams for two, three, or four variables are readily constructed (Fig. 4-22). Any number of variables may be plotted on a Veitch diagram, although the diagrams are difficult to construct and use when more than four variables are involved.

Because each variable has two possible states (true or false), the number of squares needed is the number of possible states (two) raised to a power dictated by the number of variables. Thus, for four variables the Veitch diagram must contain  $2^4$  or 16 squares. Five variables require  $2^5$  or 32 squares. An eight-variable Veitch diagram needs  $2^8$  or 256 squares—a rather unwieldy diagram. If it becomes

A

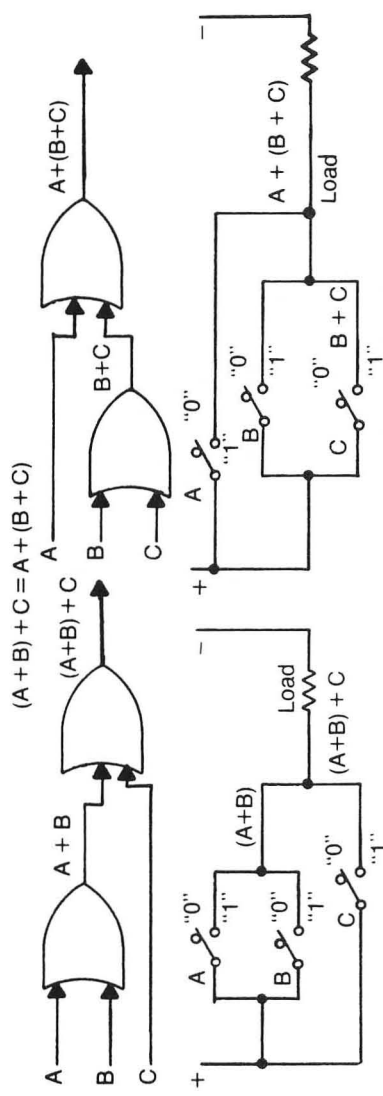
$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$



A	B	C	AB	(AB)•C	(BC)	A(BC)
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

Equal  $\rightarrow (A \cdot B) \cdot C = A \cdot (B \cdot C)$

**B**



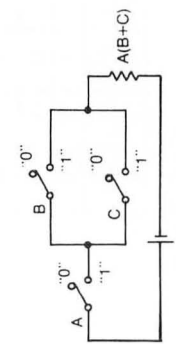
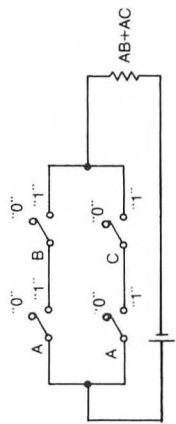
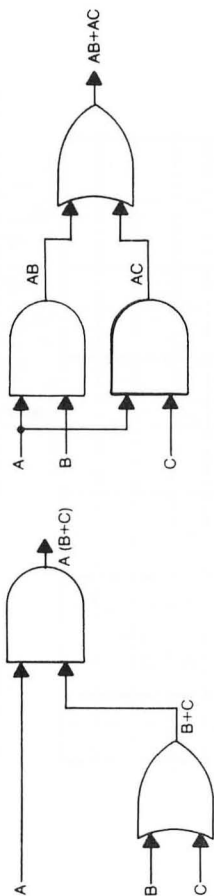
A	B	C	A+B	(A+B)+C	B+C	A+(B+C)
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

Equal
 $(A+B)+C = A+(B+C)$

Fig. 4-15. Associative law.

A

$$A(B+C) = (AB) + (AC)$$



A	B	C	AB	AC	AB+AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

A	B	C	B+C	A(B+C)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Equal  
 $A(B+C) = (AB) + (AC)$

B

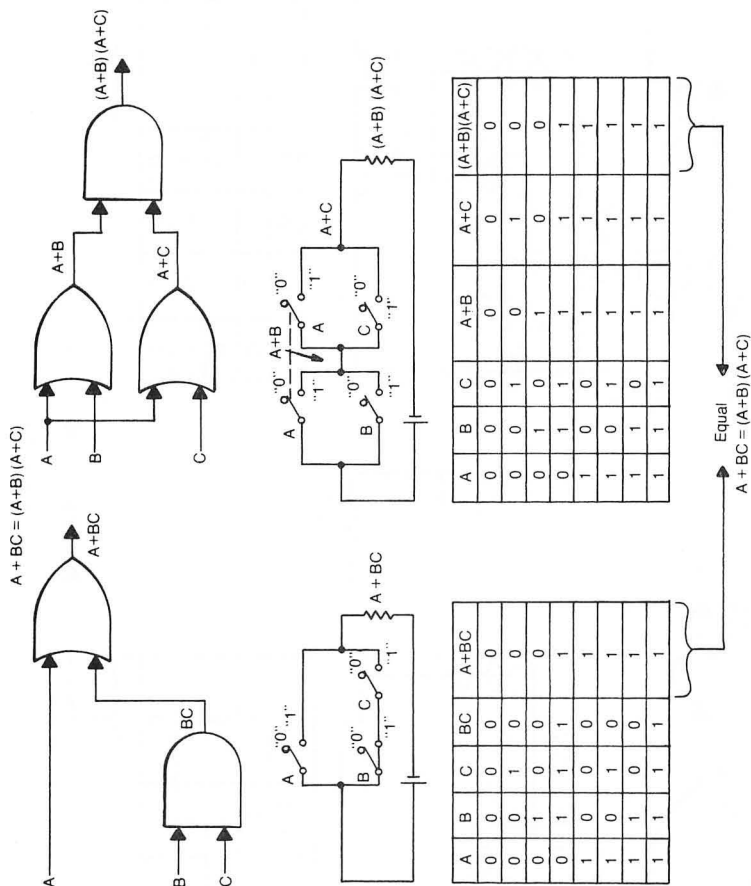
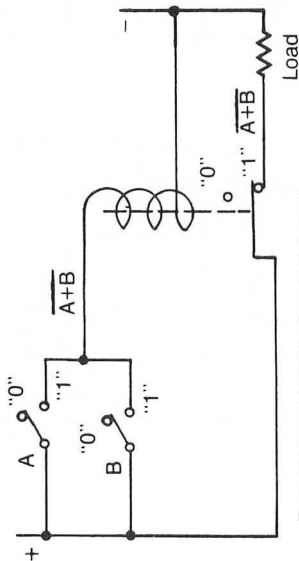
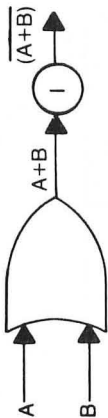


Fig. 4-16. Distributive law.

A

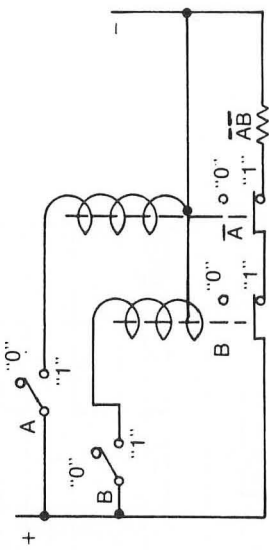
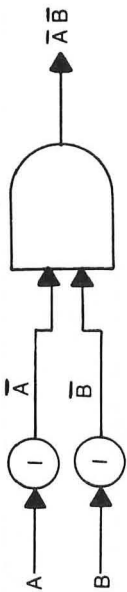
$$\overline{(A+B)} = \overline{A} \overline{B}$$



A	B	A+B	$\overline{(A+B)}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Equal

$$\overline{(A+B)} = \overline{A} \overline{B}$$



A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \overline{B}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

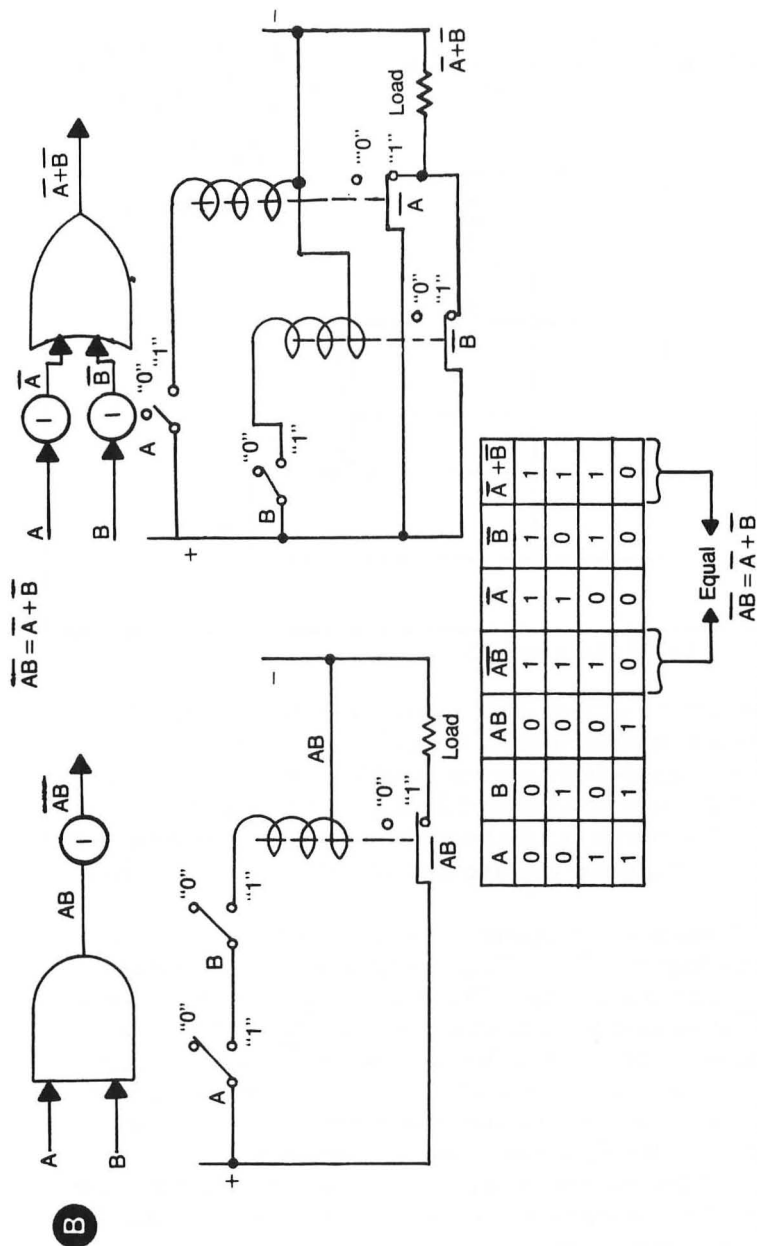


Fig. 4-17. Law of dualization (DeMorgan's Theorem).

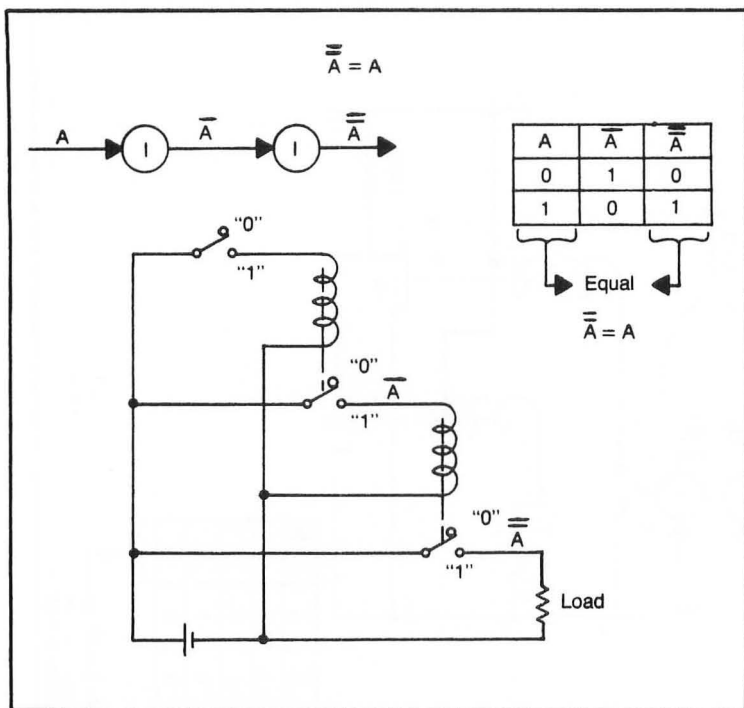


Fig. 4-18. Law of double negation.

necessary to simplify logic equations containing more than six variables, other methods of simplification should be used.

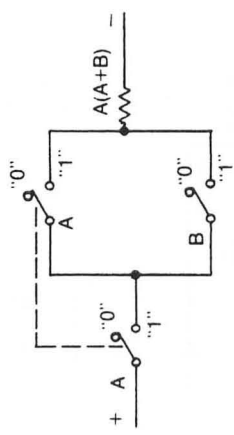
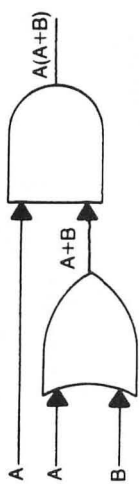
An exploded view of a four variable Veitch diagram is shown in Fig. 4-23. Note the division of the diagram into labeled columns and rows. The entries into the diagram are placed in these columns and rows in accordance with the function values for a given Boolean expression.

Looking at the square in the upper left corner of the main Veitch diagram in Fig. 4-23, and using the extensions, it is seen that it contains the variables  $AB\bar{C}\bar{D}$ ; the next lower block contains  $AB\bar{C}D$ ; the next lower block contains  $AB\bar{C}\bar{D}$ ; and the block in the lower left corner contains the variables  $AB\bar{C}\bar{D}$ . All of the squares in the diagram are similarly identified. Note that the term  $A\bar{C}$  is contained in each of the four terms just discussed. According to the distributive law, the variables  $B$  and  $D$  can be dropped, because they appear in both asserted and complemented form. Thus, the left vertical column identifies the terms  $A\bar{C}$ . This is proven by the equation:

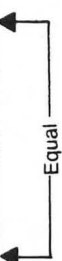


**A**

$$A(A+B) = A$$



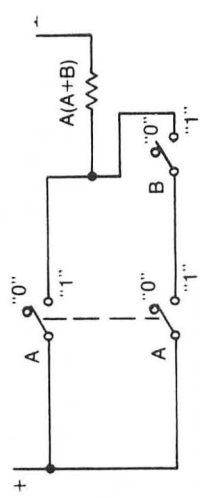
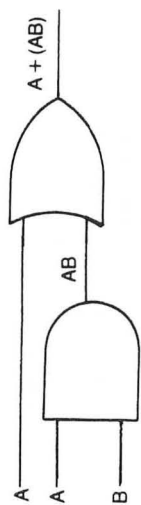
A	B	A + B	A (A + B)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1



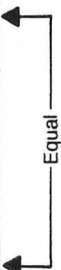
$$A(A+B) = A$$

**B**

$$A + (AB) = A$$



A	B	AB	A + (AB)
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1



$$A + (AB) = A$$

Fig. 4-19. Absorption law.

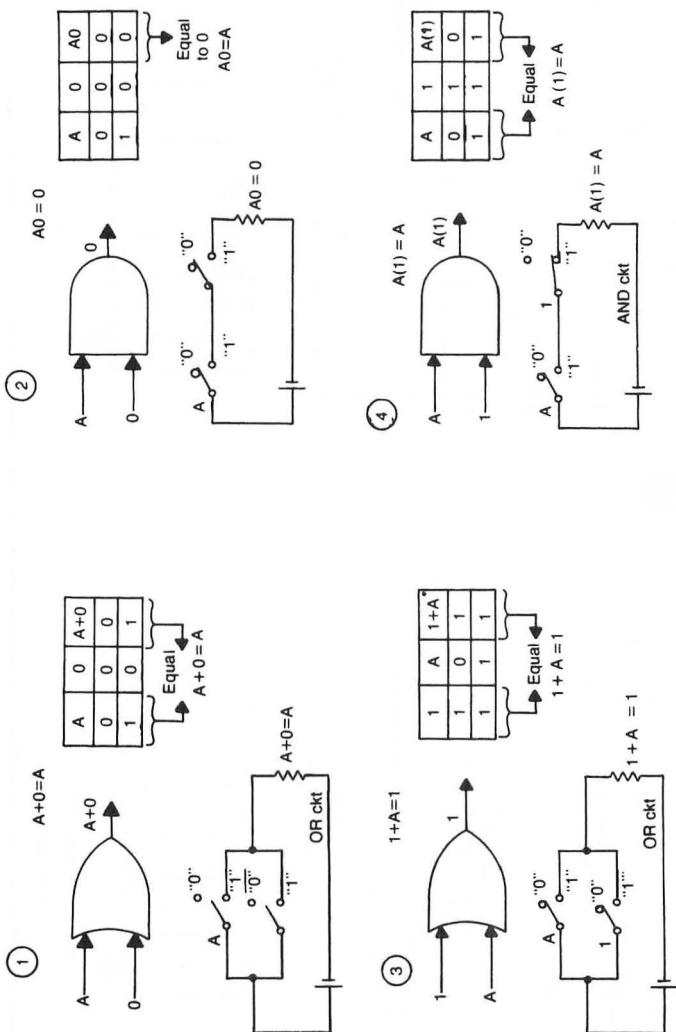


Fig. 4-20. Axiomatic expressions.

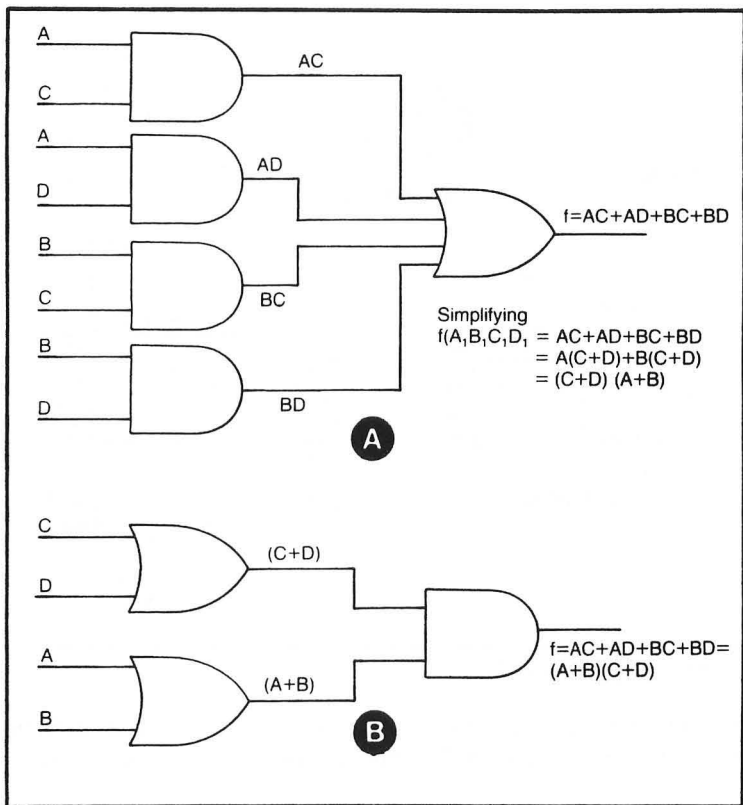


Fig. 4-21. Simplified logic circuitry resulting from simplifying logic equations.

$$\begin{aligned}
 \overline{AC} &= \overline{AC} (1) \\
 &= \overline{AC} (B + \overline{B}) \\
 &= \overline{AC} B + \overline{AC} \overline{B} \\
 &= \overline{AC} B (1) + \overline{AC} \overline{B} (1) \\
 &= \overline{AC} B (\overline{D} + D) + \overline{AC} \overline{B} (D + \overline{D}) \\
 &= \overline{AC} B \overline{D} + \overline{AC} B D + \overline{AC} \overline{B} D + \overline{AC} \overline{B} \overline{D}
 \end{aligned}$$

The final expression represents four of the maxterms of the term  $\overline{AC}$ . Also note that a two-variable term is represented by four squares. A study of the diagram will reveal that a term with one variable is represented by eight squares, a three-variable term by 2 squares, and a four-variable term by 1 square.

To illustrate the use of the Veitch diagram, the logic equation

$$f = ABC + ABD + \overline{AC} + \overline{ABCD} + \overline{AC}$$

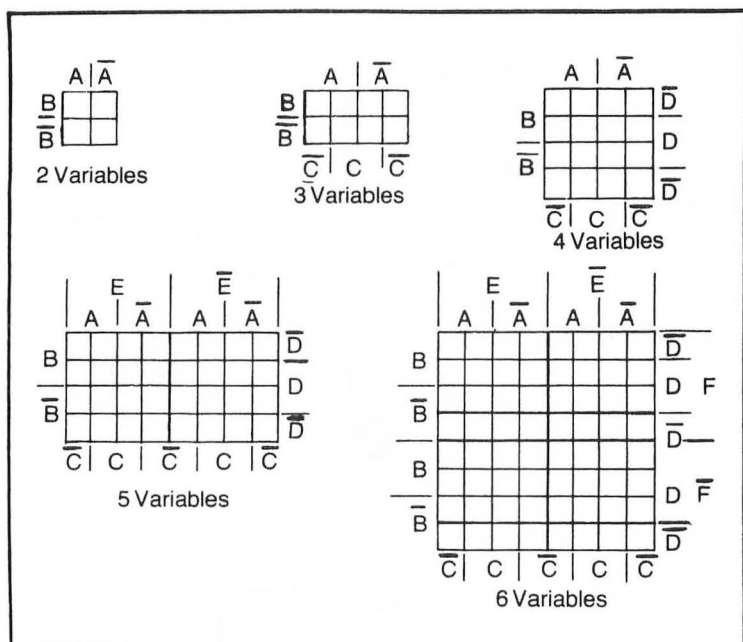


Fig. 4-22. Veitch diagrams.

will be used. Because there are four variables, a four variable Veitch diagram is needed. The step-by-step process is as follows:

**Step 1.** Draw the appropriate Veitch diagram. See Fig. 4-24.

**Step 2.** Plot the logic function on the Veitch diagram, term by term. This is accomplished by placing a "1" in each square representative of the term. (Use Fig. 4-24 to identify the squares and Fig. 4-25 to understand the plotting of the terms on the diagram.) The term  $ABC$  in the equation ( $f = ABC + ABD + AC + ABCD + AC$ ) is identified in the Veitch diagram by squares 2 and 6. The derivation is as follows:  $ABC\bar{D} + ABCD = ABC(\bar{D} + D) = ABC(1) + ABC$

The term  $ABD$ , identified by squares 1 and 2 as follows:  $AB\bar{C}\bar{D} + ABC\bar{D} + ABD(\bar{C} + C) = ABD(1) = ABD$

For the term  $AC$ , use squares 1, 5, 9, and 13:  $AB\bar{C}\bar{D} + ABC\bar{D} + \bar{A}BC\bar{D} + \bar{A}BCD = AC(\bar{D} + D) + \bar{A}BC(\bar{D} + D) = AC(1) + \bar{A}BC(1) + AC(B + \bar{B}) = AC(1) = AC$

The term  $\bar{A}BCD$ , identified by square 16, is self-explanatory. The term  $\bar{A}C$ , shown in squares 3, 7, 11, and 15, can be figured as follows:  $\bar{A}BC\bar{D} + \bar{A}BCD + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD = \bar{A}C(\bar{D} + D)$

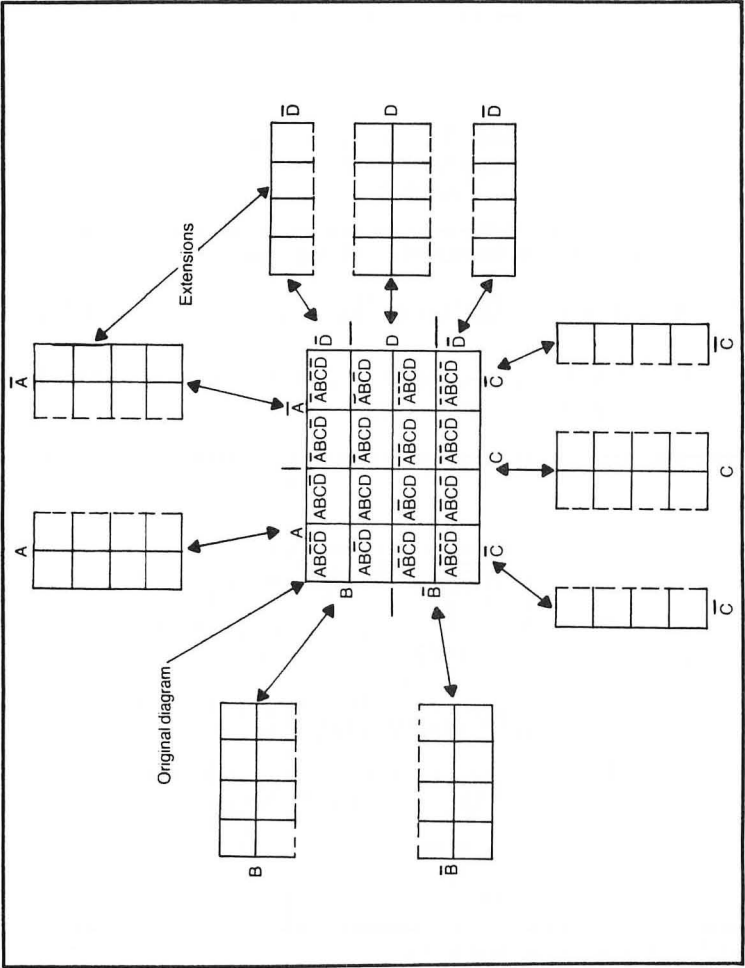


Fig. 4-23. Exploded Veitch diagrams.

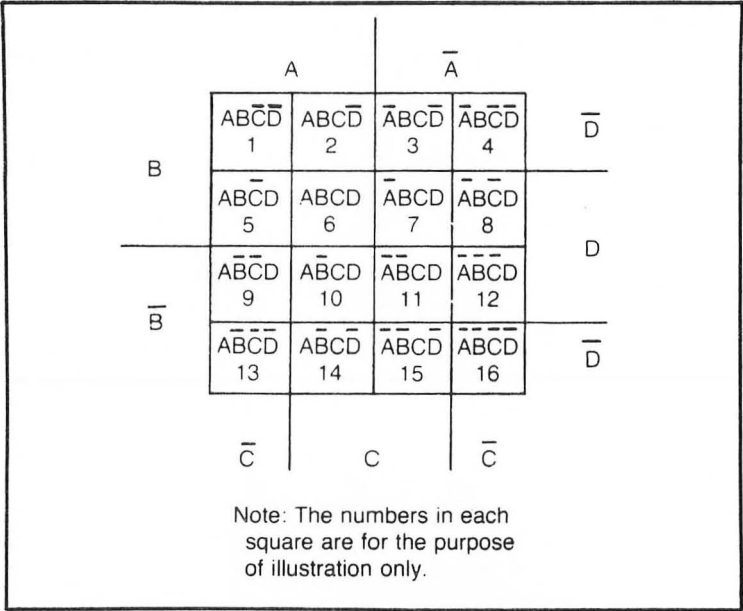


Fig. 4-24. Identifying the squares of a Veitch diagram.

$$D) + \bar{A}\bar{B}C(D + \bar{D}) = \bar{A}\bar{B}C(1) + \bar{A}\bar{B}C(1) = \bar{A}\bar{B}C(B + \bar{B}) = \bar{A}\bar{B}C(1) = \bar{A}\bar{B}C$$

**Step 3.** Obtain the simplified logic equation by using Fig. 4-26 and observing the following rules:

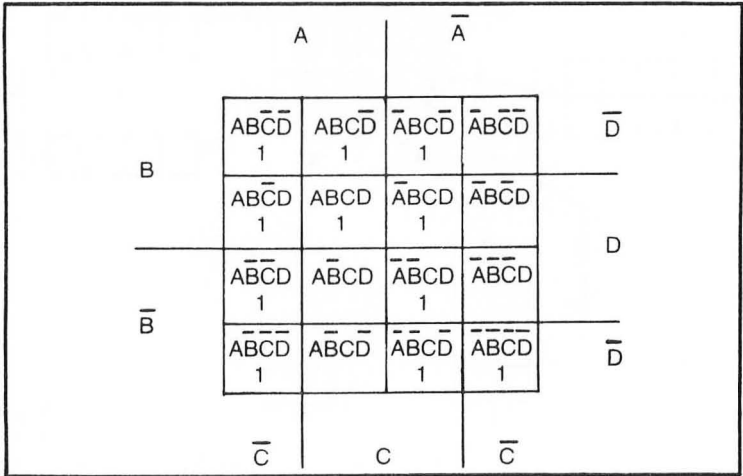


Fig. 4-25. Plotting of the logic function.

a. If 1's are located in adjacent squares or at opposite ends of any row or column, one of the variables may be dropped.

b. If 1's fill any of the following locations, two of the variables may be dropped: any row or column of squares, any block of four squares, the four end squares of any adjacent rows or columns, or the four corner squares.

c. If 1's fill any of the following locations, three of the variables may be dropped: any two adjacent rows or columns, the top and bottom rows, or the right and left columns.

d. To reduce the original equation to its simplest form, you must simplify until all 1's have been included in the final equation. The digit "1" may be used more than once, and the largest possible combination of 1's, in groups of 8, 4, 2, or as a single 1 (block), should be used.

Squares 1,5,9, and 13 are combined (Fig. 4-26) using rule (b) to yield  $A\bar{C}$  (1's in B and  $\bar{B}$  cancel).

Squares 3,7,11, and 15 are combined using rule (b) to yield  $\bar{A}C$ .

Squares 1,2,5, and 6 are combined using rule (b) to yield  $AB$ .

Squares 15 and 16 are combined using rule (a) to yield  $\bar{A}\bar{B}\bar{D}$ .

To keep track of the squares combined, draw loops around the combined squares. When you do this, the Veitch diagram takes on the appearance shown in Fig. 4-26.

All 1's have been used; therefore, a logic equation can now be written

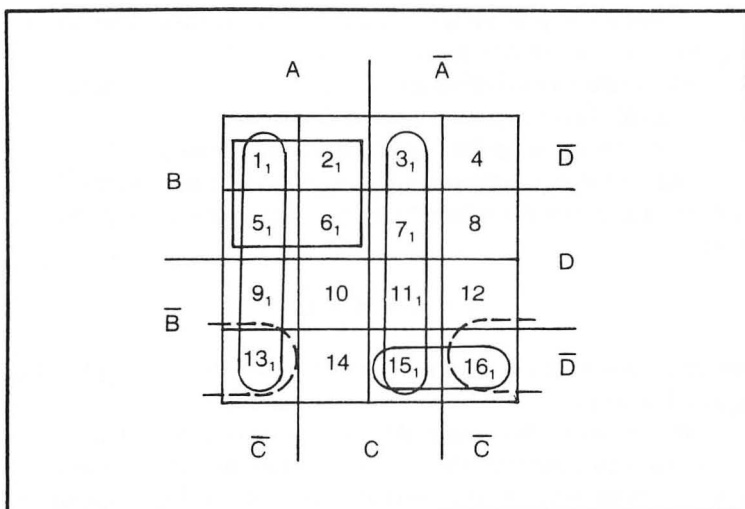


Fig. 4-26. Derivation of resultant.

	A		$\bar{A}$	
B	1	2 <sub>1</sub>	3	4
$\bar{B}$	5	6	7	8
	$\bar{C}$	C		$\bar{C}$

Fig. 4-27. Three-variable Veitch diagram showing statement True.

$$f = AB + A\bar{C} + \bar{A}C + \bar{A}\bar{B}\bar{D}$$

which agrees with the simplified logic equation obtained by the use of the simplifying theorems.

A Veitch diagram provides a convenient means of finding the complement of a logic equation. This is done by plotting the original equation on a Veitch diagram and then putting 1's on another Veitch diagram everywhere except where the original diagram has 1's. An example will illustrate the procedure.

#### EXAMPLE:

If:  $f = \overline{ABC}$

What is  $f$ ?

A three variable Veitch should give the answer. The original equation is first plotted as shown in Fig. 4-27.

On another Veitch diagram, all squares which do not have a 1 in the original diagram are assigned a 1 (Fig. 4-28).

Now the equation for  $f$  can be written. Squares 3, 4, 7, and 8 combine to form  $\bar{A}$ ; squares 5, 6, 7, and 8 combine to form  $\bar{B}$ ; and squares 1, 5, 4, and 8 combine to form  $\bar{C}$ . Therefore, the equation for  $f$  is:

$$f = \bar{A} + \bar{B} + \bar{C}$$

which agrees with the result obtained by directly applying DeMorgan's Theorem.

In the earlier discussion of logic operations, switching circuits were used to illustrate the various operations. These switching circuits were actually the mechanization of the logic operations using conventional single-pole double-throw switches. Before



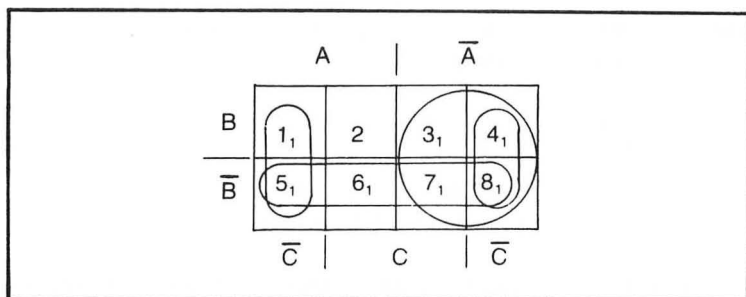


Fig. 4-28. Three-variable Veitch diagram showing statement complemented.

using the actual logic symbols, the conventional switches are again used to mechanize an equation. The equation:

$$f = AB + A\bar{C} + A\bar{C} + \bar{A}\bar{B}\bar{D}$$

will be mechanized.

It will be recalled that the AND function used a series switching circuit, and the OR function used a parallel switching circuit. Therefore, the mechanization of the above equation is as illustrated in Fig. 4-29. This diagram illustrates the AND and OR functions. The AND functions are each series-connected switch grouping of the four possible parallel paths—the OR function. The logic diagram for the above equation and mechanization are shown in Fig. 4-30, which uses the logic symbols for the AND and OR gates. A

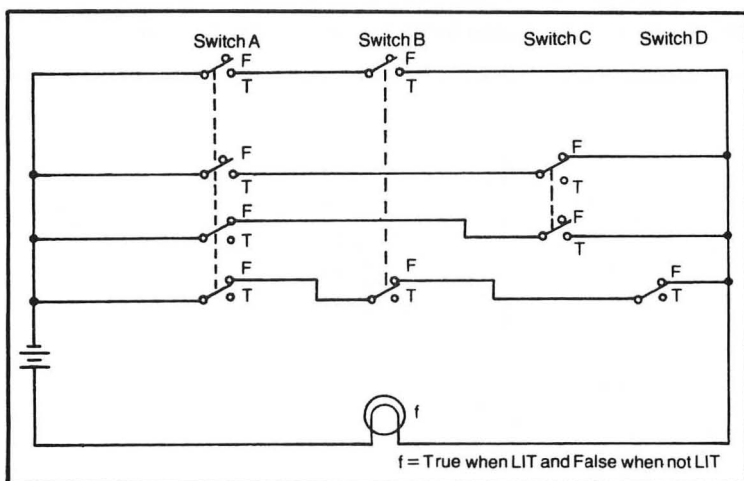


Fig. 4-29. Mechanization of a logic equation.

logic equation can be always mechanized by a switching network. This involves the following four steps:

1. Construct a truth table.
2. Write the logic equation.
3. Simplify the equation, if possible.
4. Draw the required switching network.

You are encouraged to apply these four steps to several hypothetical problems.

Consider the following equations:

$$1. f = \overline{A}B + A\overline{B} + \overline{A}\overline{B}$$

$$2. f = A + \overline{B}$$

Equation 1 is the sum of three Boolean terms, each of which is the product of two variables (A and B). Each variable is represented in either its true or complemented form. Equation 2 represents the sum of the two variables A and B when B is complemented. Equation 1 is a minterm expression of the two variables, and equation 2 is a maxterm expression of these variables.

In general, a minterm expression of n variables is defined as the product of these n variables where each variable is expressed in either its true or complemented form. A maxterm of n variables is the sum of these n variables where each variable is added in either its true or complemented form. Consequently, there are four min-

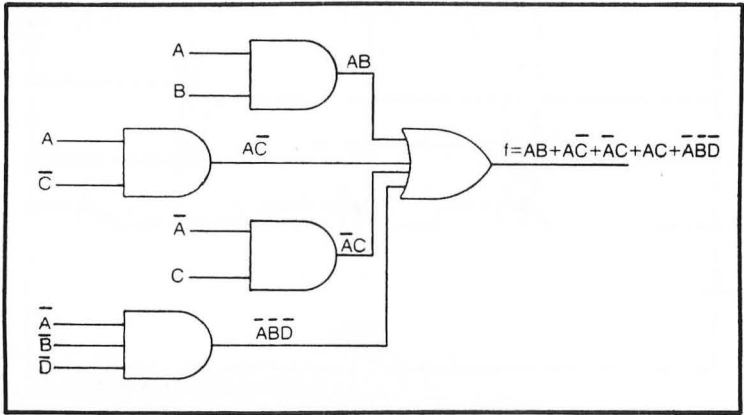


Fig. 4-30. Mechanization of a logic equation using logic symbols.

	Minterm		Maxterm
1	$\overline{\overline{A}}\overline{\overline{B}}\overline{\overline{C}}$	16	$\overline{\overline{A}}+\overline{\overline{B}}+\overline{\overline{C}}$
3	$\overline{\overline{A}}\overline{\overline{B}}C$	14	$\overline{\overline{A}}+\overline{\overline{B}}+C$
5	$\overline{\overline{A}}\overline{\overline{B}}\overline{\overline{C}}$	12	$\overline{\overline{A}}+\overline{\overline{B}}+\overline{\overline{C}}$
7	$\overline{\overline{A}}\overline{\overline{B}}C$	10	$\overline{\overline{A}}+\overline{\overline{B}}+C$
9	$\overline{\overline{A}}\overline{\overline{B}}C$	8	$\overline{\overline{A}}+\overline{\overline{B}}+C$
11	$\overline{\overline{A}}\overline{\overline{B}}C$	6	$\overline{\overline{A}}+\overline{\overline{B}}+C$
13	$\overline{\overline{A}}\overline{\overline{B}}C$	4	$\overline{\overline{A}}+\overline{\overline{B}}+C$
15	$\overline{\overline{A}}\overline{\overline{B}}C$	2	$\overline{\overline{A}}+\overline{\overline{B}}+C$

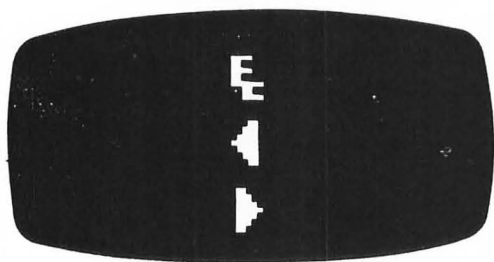
Fig. 4-31. Minterms and maxterms of variables A, B, and C.

terms of two variables, and they are  $(AB, \overline{AB}, A\overline{B}$  AND  $A\overline{B})$ . Likewise, there are four maxterms of two variables. They are  $(A + B), (A + \overline{B}), (\overline{A} + B)$  and  $(\overline{A} + \overline{B})$ .

There are eight minterms and eight maxterms of three variables as shown in Fig. 4-31. As might be expected, there are  $2^n$  minterms and  $2^n$  maxterms when  $n$  variables are considered.

Observe that the minterm identified by an odd number in the upper left corner of each block in the left column relates to the next higher even number in the right column in accordance with DeMorgan's Theorem. For example, the maxterm in block #2 is the complement of the minterm in block #1.

# Chapter 5



## Introduction to Programming

The process of writing instructions to control the operation of a computer is called programming. Most of the remaining chapters in this book will deal with programming in one form or another so we will only briefly describe the basic technique at this point.

In general, computer programming refers to the analysis and planning of a problem solution. The phase of instruction writing is referred to as coding. Here are a few of the concepts involved in programming.

**Binary Operation:** The programming process will be described in terms of the most common form of digital computer. This machine operates internally entirely in binary (base 2) arithmetic logic, and is arranged so that information is stored and accessed in units termed words. The ATARI uses words that are 8 bits in size.

**Decimal Operation:** The other broad class of computers operates logically in the decimal system (although decimal digits are fabricated within the machine by combinations of bits) and stores and accesses its information in units of individual characters such as digits or letters. This machine is referred to as a character-addressable or variable-word-length computer.

**Notation:** The ATARI operates internally using the binary number system. Skilled programmers often work using the octal (base 8), or hexadecimal (base 16) systems. These systems are simply conveniences for programmers to use while they work. However, the discussion here will be in terms of decimal values to

simplify understanding of computer principles. It must be kept in mind that the basic nature of the machine is binary.

**Number Operations:** The purpose of a computer is to manipulate information that is stored within the machine in the form of binary numbers. These numbers can be treated as symbols and can represent alphabetical information. They can also be treated as numbers and can be used arithmetically. Each word contains one number and its associated algebraic sign (positive or negative). The instructions which dictate the operations to be performed on such numbers (data) are also numbers and are also as a single word in the same physical medium as the data. The stored programming concept presumes that there is no physical difference between these two types of numbers and that the instruction numbers may also be manipulated, in the proper context, as data.

**Addresses:** Each word in storage is assigned an address in order to refer to it, in a manner analogous to postal addresses. The addressing scheme is part of the hardware of the machine, and is wired in permanently. Word addresses range from zero to the storage size of the machine. Any word may be loaded with any desired information. Some of this information is the data of the problem being processed; the rest is the instructions to be executed. The machine is designed basically to execute instructions in the order in which they are stored, advancing sequentially through the instruction words at addresses that increase by one.

**Instruction Format:** Each instruction contains two basic parts: (1) a coded number that dictates what operation is to be performed and (2), the address of the word (data) on which to perform that operation.

Programming for a digital computer could be done in binary, but becomes increasingly difficult as programs get longer. For this reason, programmers prefer to work in languages that are at a higher level. In higher-level languages, mnemonic operation codes are used in place of numbers, and all absolute machine addresses are replaced by symbols. These alterations greatly speed up the work of the programmer. High-level languages permit a format that fits the problem rather than the machine. BASIC is one such language that is readily adapted to both home and business applications.

## **BASIC**

BASIC means *Beginner's All-Purpose Symbolic Instruction Code*. It is a high-level language that a beginning programmer can

understand with relative ease. An *interpreter* that changes the instructions in BASIC to terms that can be used by the machine must be present in the computer when a BASIC program is used. BASIC was developed at Dartmouth College in the sixties for educational purposes. It is, in fact, probably the simplest higher-level language available at this time.

BASIC is widely implemented on microcomputers like the ATARI 800, and it is also available on most large scale computers.

## THE PROGRAMMING PROCESS

Programming is a creative process because it is primarily a problem-solving job. You, the programmer must select the best or most effective and efficient solution for a given problem. Once a solution is developed, you must then code it into a language the computer understands, so that the computer can help solve the problem.

Before beginning to write a program to solve a given problem, there are some basic considerations with which you should become familiar. In general, there are four objectives in programming:

1. Economy of storage
2. Speed of computation
3. Accuracy
4. Simplicity

With these four basic objectives in mind, let's see what steps are involved when programming a computer.

1. Accurately define the problem—exactly.
2. Propose a feasible and economical solution.
3. Redefine the problem in terms of what the computer is expected to do.
4. Prepare a flowchart of the computer solution.
5. Code the solution.
6. *Debug* (remove errors from) the program.
7. Document the program.

Before you can solve a problem, you must understand the problem. First make a list of all available data. Then define the problem in terms that are understandable terms both to yourself and others who might be involved with the problem. Finally, propose a method of solving the problem. If there is more than one possible solution—and there usually is—choose the one that best fulfills the

four basic objectives of programming, especially simplicity.

When you define a problem and then propose a feasible solution to it, you are developing what is called an algorithm, or a step-by-step solution to a specific problem. Once this is accomplished, you must tailor the solution to fit the computer's capabilities, or the solution will be useless. You must define both the problem and solution in terms of what the computer can do.

Next, you should draw a flowchart of the solution. Basically, a flowchart is the algorithm illustrated in graphical form. See Chapter 13 for more details. After the solution is flowcharted, it can be coded into a language that the computer understands. You must then check the program for errors and finally, document it.

## **LET'S WRITE A PROGRAM**

We could continue on the theory of program writing for several more pages, but to get you acquainted with the practice, let's jump right in and try writing a simple program. If this is your first attempt at such an undertaking, there will still be a few hazy areas once we are finished, but you should have a good basic knowledge which will enable you to attempt the more advanced problems later on in the book.

Remember, to write a computer program, you must have the following:

1. A precise definition of what the program is to accomplish.
2. A detailed, step-by-step plan of how the program goals will be achieved.
3. Knowledge of the programming statements can be used to implement the step-by-step plan.

### **Define precisely what the program is to accomplish:**

You are more than likely familiar with the Fahrenheit temperature system. You have also probably heard of the Celsius temperature system that is supposed to eventually replace the Fahrenheit system. So let's write a temperature conversion program. Here's exactly what the program is to accomplish:

This program is to provide a fast and easy means of converting between the Celsius and Fahrenheit temperature systems. The direction of the conversion is to be specified; then the temperature that is to be converted must be entered. The converted temperature is then displayed or printed.

This precise definition of what the program is to accomplish is the first and most important part of program writing. With this out of the way, we can continue to the second step.

**Make a step-by-step plan of how the program goals will be achieved:** There will usually be more than one solution to a problem, so the exact method of developing the step-by-step plan will differ with each programmer. In most cases, however, it is best to rely on the technique known as flowcharting. A flowchart is simply a set of symbols that indicate the steps of program execution. Each block in a flowchart usually contains one and only one instruction. The standard flowcharting symbols appear in Fig. 5-1.

The terminal symbol is used to indicate the beginning and ending points of a program; words like “start,” “end,” or “stop” are normally written inside it. The input/output symbol is used to indicate an input or output operation. The decision symbol indicates a decision or yes/no question, with the nature of the decision normally written inside the symbol. As might be imagined, the yes and no paths provide two directions for the program to continue in, depending upon the result of the decision.

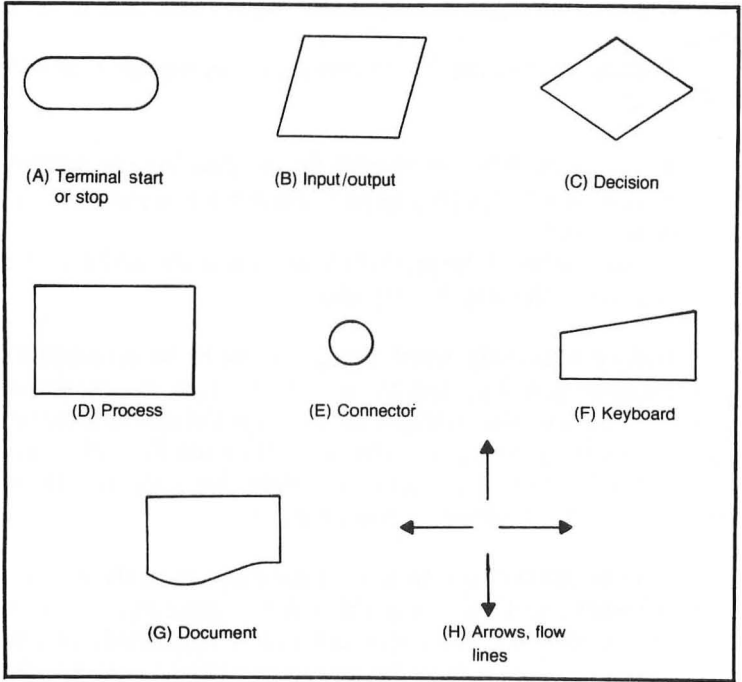


Fig. 5-1. Flowcharting symbols.



The process symbol is a rectangle. It indicates operations such as add, subtract, load accumulator, or store. The circle is used to refer from one point to another when it is difficult or impossible to physically connect them with a line. For example, corresponding letters placed in these circles could show the continuation of a program from one sheet of paper to another.

The keyboard symbol is used to represent an output to a video terminal or input from a keyboard. The document symbol generally represents an output on a printer. Finally, the arrowheads and the flow lines depict the relationship between symbols and identify operational sequences.

To write a flowchart for our sample program, we must take the description of exactly what the program is to accomplish and provide the appropriate flowchart symbol for each step.

First, each program must have a beginning point.

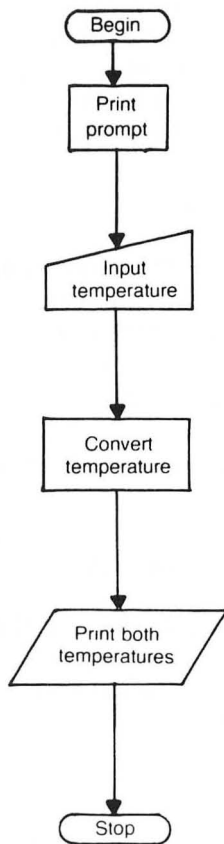
Next a prompt must indicate what information the user should input.

Now, the description says we will have to input a temperature in either Celsius or Fahrenheit. We can change this each time the program is run, so the information should come from the keyboard.

Now the computer must perform an operation (convert the temperature in one system to the other).

Now the results of the conversion (the output) should be printed.

All programs must have an Ending Point.



**Implement the step-by-step plan with programming statements:** Before we can actually write the program, we must have an understanding of standard programming statements. The BASIC programming statements used in this program are shown in Table 5-1. Let's apply these statements to accomplish the steps shown in the program flowchart.

The first step in the flowchart is "Begin". Since we do not need a program line to "begin", we will take this opportunity to write a REMark statement that gives the title of the program:

```
10 REM **CELSIUS TO FAHRENHEIT**  
20 PRINT "CELSIUS TEMP"
```

The second statement PRINT "CELSIUS TEMP", causes the computer to display the question, CELSIUS TEMP? The second step in the flowchart calls for an input; in this case, the temperature in the Celsius. So the response to the next line of the program (30) will come from the keyboard. We need to select a name for the variable which will receive the data. How about "C" for Celsius?

```
30 INPUT C
```

The next step in the flowchart calls for a computer operation: the Celsius temperature must be converted to Fahrenheit. To convert the Celsius temperature to Fahrenheit, the Celsius temperature is multiplied by 1.8 and then 32 degrees is added to the results. So our next statement will be:

```
40 LET F = 1.8*C+32
```

The next step in the flowchart calls for printing the results of the computation.

```
50 PRINT C; "C EQUALS ";F;"F"
```

The last step on the flowchart is the end of the program so we'll write

```
60 END
```

Here is the initial version of our program

**Table 5-1. BASIC Programming Statements.**

Atari Programming Statements

Let	Assigns names and values to variables.
PRINT "string literal"	Prints on paper or displays on screen the exact characters enclosed by quotation marks.
PRINT variable name	Prints or displays numerical content of variable.
PRINT expression	Computes mathematical expression; then prints or displays the result.
PRINT item; item	List of items printed or displayed on same line without added spaces.
PRINT item , item	List of items printed or displayed on same line aligned into columns
REM	Remarks can be included in program to document or explain. Ignored by computer.
Goto	Alters order of program execution.
If...then	Makes decisions based on relational tests.
Read...data	Supplies data items from program-resident list.
Input	Supplies data items from keyboard.
Stop	Stops program execution.
Deluxe Programming Statement	
On...goto	Program branches made to computed line number.

```

10 REM **CELSIUS TO FAHRENHEIT**
20 PRINT "CELSIUS TEMP"
30 INPUT C
40 LET F = 1.8*C + 32
50 PRINT C; "C EQUALS ";F; "F"
60 END

```

And here's how the program will look when executed:

```

CELSIUS TEMP?
15
15 C EQUALS 59F

```

To satisfy our initial program requirements, we must write another program to provide a means to convert Fahrenheit to Celsius. In general, we merely reverse the operations of the first part of the program to come up with the following scheme.

```

10 REM **FAHRENHEIT TO CELSIUS**
20 PRINT "FAHRENHEIT TEMP";
30 INPUT F
40 LET C = 5/9*(F-32)
50 PRINT F; "F EQUALS " ;C;"C"
60 END

```

Assuming we have a Fahrenheit temperature of 59 degrees, what is its Celsius equivalent?  
Here's how the program will look when executed.

```

FAHRENHEIT TEMP?    (The computer asks)
59                  (You type this in)

```

The computer will print or display the following:

```

59 F EQUALS 15 C

```

## LINE NUMBERS

From the previous discussion, we know that a BASIC program is composed of a series of statements which manipulate data. Each BASIC statement occupies one line. Each line is identified by a number which appears at the beginning of the line. Obviously, no two lines may use the same number. In most cases, the computer

executes the statements in the order of their line number, and these line numbers are normally increments of ten; that is, 10, 20, 30, etc. This gives the programmer nine numbers between statements to add other statements later to expand the program if necessary.

## BASIC STATEMENTS

Earlier in this chapter you were introduced to BASIC statements, but since these must be thoroughly understood to program in BASIC, let's review these statements and go into more detail. There aren't really very many to learn, so make sure you know and understand each before continuing with programming.

**Let:** The let or replacement statement is used to assign a value to a variable name. For example, in our temperature conversion program, `40 LET C = 5/9*(F-32)` assigns C the value of 5/9 times the Fahrenheit temperature minus 32 degrees.

Successive let statements can be used to place a number of constants into different variables. For example, suppose we wish to place the four constants 21, 32, 43, -12 in variables A, B, C, and D respectively. Four let statements could be written as follows:

```
10 LET A = 21
20 LET B = 32
30 LET C = 43
40 LET D = -12
```

The four constants 21, 32, 43, and -12 are then assigned to the four variables A, B, C, and D.

**Read and Data Statements:** While the let statement allows for the placement of a constant into a variable, its use becomes unwieldy if a large number of constants are to be entered. In the above example, four statements were required to enter four constants. The read and data statements are used to overcome this difficulty. To assign the four constants in the above example, only two statements are required. One statement identifies the constants (data). It would be written:

```
10 DATA 21, 32, 43, -12
```

The read statement assigns these constants to the variables. It would be written:

```
20 READ A,B,C,D
```

The first constant is assigned to the first variable listed, the second constant to the second variable, and so forth. Thus, with only two statements we could assign dozens of constants to corresponding variables. The read and data statements are the most common way of entering constants into the computer.

**Print Statement:** Once the program has finished, we need a means of displaying the results. The print statement is used for this purpose. It consists of the word print followed by a list of the data and variables to be put on the screen. For example, if we wished to print out the values of variables A,B,C, and D, we would write:

```
20 PRINT A,B,C,D
```

When this statement is executed, the constants stored in these locations are printed.

The print statement also allows a message to be printed. The message is enclosed in quotation marks. For example, the statement:

```
30 PRINT "END OF DATA"
```

causes the words END OF DATA to be printed. Notice the difference between the two statements PRINT X and PRINT "X". In the first statement, the value assigned to the variable X is printed. In the second statement, the message X is printed. To illustrate how this can be used, consider the program:

```
50 LET X = 3  
60 PRINT "X=",X
```

Statement number 50 assigns X the value of 3. Statement 60 causes the message "X=" to be printed. It is followed by the value of X. This gives a printout of

```
X = 3
```

**If...then statement:** One of the most powerful control statements in BASIC is the if...then statement. It is a decision-making statement that allows the computer to choose alternative courses of action, depending upon the relative size of two constants.

**Table 5-2. Symbols Used in the BASIC Programming Language.**

Symbol	Meaning
=	is equal to
<	is less than
>	is greater than
< =	is less than or equal to
> =	is greater than or equal to
< >	is not equal to

The two constants can be compared in any of six different ways. The symbol explaining the relationship of the two constants are shown in Table 5-2.

The if...then statement takes the form:

```
10 IF X > 0 THEN 30
```

This says that if X is greater than 0, execute statement 30. If X is not greater than 0, then the next statement in the sequence is executed. As you can see, this type of statement is a conditional branch which depends on the value of X.

**Goto Statement:** The goto statement is the unconditional branch statement. It consists of the word goto followed by a line number. For example, GOTO 160 causes line 160 to be executed next rather than the line that is next in sequence.

**On...goto Statement:** This statement is a conditional multiple-branch statement. It is also known as a computed goto statement. The format is:

```
10 ON X GOTO 20,30,40
```

The integer portion of X is evaluated. If X = 1, the program branches to statement 20. If X = 2, the program branches to statement 30. If X = 3, the program branches to statement 40. Although only three possible branches are shown here, the statement can contain as many possibilities as needed. Also, the X may be a formula or expression rather than a single variable. An expression such as X = A - B would be evaluated first; then the outcome would be used to determine the branch. Noninteger values of X will be truncated to their integer values before being used to determine the branch. As you can see, this type of statement can become very complex. However, when properly used, it can do the same job as many if...then statements.

**For...To and Next Statement:** These statements are used to control the repeated execution of a program loop. When using these statements, the program loop begins with the for...to statement and ends with a next statement. The format of the for...to statement is

```
15 FOR X = 1 to 100
```

When this statement is encountered, a counter, X, is set to the first value on the right side of the equal sign. In the example above, X is initially set to 1. The program then executes the statements in the loop. If the variable X is encountered in any of the statements in the loop, it is assumed to have the value of 1.

The last statement in the loop is the next statement. Its format is:

```
NEXT X
```

This informs the computer that the first pass through the loop is complete and that X should be set to its next sequential value. In this case, X is incremented by 1 to the value 2 and the loop is repeated. The loop is repeated again and again with X being set to every value from 1 to 100. Each time the next X statement is executed, the new value of X is compared with the final value of 100. This continues until X has assumed all values called for in the for...to statement. The loop is repeated for the last time when X is set to 100. At this time, the program leaves the loop and executes the first statement following the next.

As a simple example, consider the loop below

```
10 FOR X = 1 TO 5  
20 PRINT X  
30 NEXT X
```

The computer will set  $X = 1$ , then print it. X will then be incremented and the loop repeated. The program above will produce the following output:

```
1  
2  
3  
4  
5
```



In our example, X was set to all values for 1 to 5. The for...to statement could just as easily have called for all values from 1 to 1000 or from 13 to 20, etc. The variable which assumes these values need not be X. Any variable could have been assigned. Most of the time when a for...to statement is written, a *step* size of 1 is used. In our example, X was stepped (incremented) by 1 each time through the loop. However, other size steps can be specified by the for...to statement. For example the statement:

```
FOR N = 2 TO 1000 STEP 2
```

specifies that the variable N will be assigned the values 2,4,6,8,...1000. The for...to statement can also be told to count backwards by using the following format:

```
FOR X = 1000 TO 1 STEP-1
```

In other words, X is decremented.

**Subscripted Variables:** It is often convenient to use subscripted variables when programming with BASIC. For example, if we have a list of 20 numbers to be added, we can call the entire list by a single variable name. If we call the list A, the first number on the list could be  $A_1$ ; the second,  $A_2$ ; the third,  $A_3$ ; and so forth. When entering data from a video terminal or I/O typewriter, the subscript must be typed on the same line as the variable. For this reason, subscripts are written in parentheses. For example,  $A_1$  is written A(1),  $A_2$  is written A(2), etc. Be careful not to confuse subscripted variables such as X(1) with simple variables composed of a letter and a numeral such as X1.

In the terminology of BASIC, the list of 20 numbers described above is called an array. We arbitrarily called the array by the variable name A. In BASIC, an array name must consist of a single letter. Since there are only 26 letters, we can use no more than 26 arrays in a single program. The list of 20 numbers described above is called a one-dimensional array. The 20 numbers in the array are known by the subscripted variables A(1) through A(20). An array can sometimes have two dimensions. For example, consider the numbers shown in Table 5-3. This table contains 20 numbers just like the list discussed earlier. However, the table is divided into 4 columns of 5 numbers each. Thus, we call this a two-dimensional array or matrix. The numbers in this type of array are described by double subscripted variables. If the array is A, the number in the first

**Table 5-3. A Two-Dimensional Array.**

First Subscript (rows)	Second Subscript (columns)			
	(1)	(2)	(3)	(4)
(1)	31.7	79.6	16.8	11.5
(2)	19.9	37.3	87.2	53.6
(3)	10.3	94.0	83.2	44.4
(4)	76.1	19.9	49.4	29.1
(5)	01.7	21.7	57.9	37.3

row and first column is A(1,1). By the same token, the third number in the second column is A(3,2). Arrays are useful when large amounts of data must be handled nonsequentially or when alternations of data must be achieved.

**Dim Statement:** The dim or dimension statement is used to deal with lists and two-dimensional arrays in BASIC. It allocates storage and assigns names to the lists and tables to be used. The dim statement has two formats:

```
30 DIM A(6)
40 DIM F(10,4)
```

The first statement defines a list called A and allocates 6 memory locations for storage. The second statement defines a table (a two-dimensional array) called F with 10 rows and 4 columns. Several lists or arrays may be defined with one dim statement by simply including the definitions on the same line separated by a comma.

**Gosub and Return Statements:** In BASIC, as in most languages, subroutines can be used. An often needed computation can be written in BASIC, then used by the main program as often as needed. The gosub statement causes the main program to branch to the subroutine. A line number following the word gosub tells where the subroutine is located. For example, the statement gosub 205 tells the program to branch to line 205 and execute the statements it finds there. The end of the subroutine is signified by a return statement, which sends execution of the program to return to the main program. The return causes the statement following the one containing the gosub to be executed next.

**List Command:** List is generally used as a command, not as a statement within a program. It causes the program in the com-

puter to be displayed in its entirety. This allows you to review, edit, or modify the program conveniently without retyping it.

**REM Statement:** The remark statement lets the programmer insert notes, comments, or messages of any kind into the program. The remarks do not affect the program itself. They simply allow you to tell what the program is, how it is used, and how to use it. For example, 10 REM PROGRAM FOR COMPUTING GRADE AVERAGE will cause the statement PROGRAM FOR COMPUTING GRADE AVERAGE to be printed when the program is listed.

**Input Statement:** We have seen that data can be assigned to variables by using the let statement or by using the read and data statements in pairs. There is also a third method of assigning values. It involves the use of the input statement. The format of the input statement is

```
50 INPUT A,B,C
```

If this input statement is used in a program, the computer executes the program in the normal fashion until this statement is encountered. When the computer finds the input statement, it will display a question mark on the terminal. After displaying this question mark, the program cannot proceed until a reply is received from the user. The user sees the question mark and types the values he wishes to assign to the variables A, B, and C. The computer then accepts the three values, assigns them to variables A, B, and C, and continues the execution of the program.

The input statement can specify as many or as few variables as needed. However, when the computer types the question mark, the program must respond by typing as many constants as there are variables in the input statement.

It is often convenient to have several different input statements scattered through a program. Each input statement can specify a different combination of variables. In this case, it would be very easy for the programmer to forget which input statement calls for which variable. For this reason, programmers usually adopt the technique of placing a print statement just before the input statement. For example, in the following program, line 20 causes a message to be printed which clarifies the question mark produced by line 30. The program looks like this:

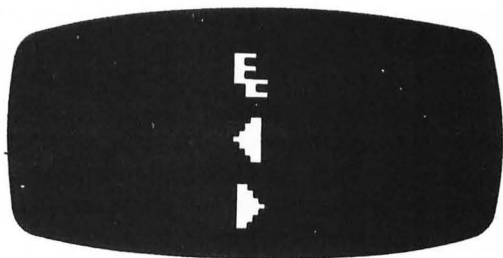
```
20 PRINT "WHAT ARE THE VALUES OF A,B,C,X, AND Y";  
30 INPUT A,B,C,X,Y
```

Without the print statement the computer simply prints a question mark. However, with the print statement the computer prints:

WHAT ARE THE VALUES OF A,B,C,X, AND Y?

By requiring a reply from the programmer, the input statement allows the “real-time” input of data. When used with an appropriate print statement, it also allows two-way communication between the programmer and the computer. In fact, if enough print and input statements are carefully used, it can appear that the computer and the programmer are carrying on an intelligent conversation.

**End Statement:** The end statement is important because it must make up the last statement in any program. It consists simply of a line number followed by the word end. When the computer encounters this statement, it stops execution of the program.



### Introduction to the ATARI 800

The ATARI 800 is a series of components which function together with a television set as a single system. The basic system includes:

- \* Computer Console
- \* TV Switch Box
- \* Ac Power Adapter
- \* Program Recorder and Power Cord
- \* 2 Instruction Books:
  - Operators Manual
  - ATARI BASIC
- \* 2 Cartridges:
  - ATARI Educational System
  - ATARI BASIC LANGUAGE

The software components are the operating system software, the programs inside the ATARI Educational System, the ATARI BASIC Language cartridges and, of course, the contents of the two books.

The basic system may be enhanced in several ways: with controllers; 8 and 16K RAM memory modules; a printer; floppy disk drives, and cartridge, disk, and cassette software.

Inside the ATARI 800 Console are the central processing unit (CPU) and the memory bank containing the operating-system read-only-memory (10K ROM) module and 8K (8 thousand characters or "bytes") of user programmable random access memory (8K RAM), and two expansion sockets for additional RAM memory

modules. The Console also holds the keyboard, cartridge slots, controller jacks, and a serial I/O Port for connecting to peripheral components.

The TV switch box allows you to change from regular TV reception to computer display by moving the sliding switch on the box. The ac power adapter plugs into a normal wall socket and converts it to the low voltage used by the ATARI 800.

The ATARI program recorder provides software storage in computer readable form. You may purchase preprogrammed cassettes from your ATARI retailer or other software and computer dealers and you may use any blank, high-quality audio cassette tape to save programs you write yourself.

## **UNDERSTANDING SOFTWARE**

The ATARI 800 Basic System is very much like an empty sheet of paper. It has the potential for an unlimited number of applications, but that potential remains dormant until you add the software.

The software that accompanies the ATARI computer consists of programs considerably more complex than those you studied in the preceding chapter. The foundation programs are supplied in the operating system ROM module. They activate the keyboard and the screen display so that you can create pictures and text one screenful at a time. These programs control the flow of all information within the computer.

Usually you add a second level of software to the ATARI 800. This can be done by inserting a cartridge into the cartridge slot. This software transforms the ATARI 800 into a special purpose machine for playing a game, presenting educational material, manipulating information or entering programs through the keyboard. The program recorder and optional floppy disk drive provide additional methods for loading programs into the computer.

ATARI application cartridges contain programs that are permanently recorded in a ROM within the cartridge. They control the computer in machine language, the most intricate level on which to manipulate the computer. These cartridge programs produce full-color, animated displays and complex electronic decision-making. Many entertaining game cartridges are available, and each is designed to be easy to learn but difficult to master.

Other ATARI Cartridges have a more serious purpose. They are tools for increasing your speed and accuracy in handling words and numbers. ATARI programmers identify and analyze problems of interest such as checkbook or mail list management. They then

design a generalized solution to each problem, program that solution in machine language, and record it in the cartridge. When you insert the cartridge, the ATARI 800 repeats this preprogrammed solution, substituting your data from the keyboard into its equations. Although all cartridges operate in the same general fashion, each cartridge causes the ATARI 800 to use the screen display, keyboard, and/or controllers in a different way. You will need to read the instructions which accompany each cartridge for specific details.

## KEYBOARD

The ATARI 800 Keyboard (Fig. 6-1) has alphabetic, numeric, graphic and screen editing functions which are detailed in the paragraphs to follow. Most keystrokes produce a visible change on the display screen. However, there are a few keys which are only used in combination with others. To investigate the effects of each key, power up your ATARI 800 without a cartridge in the cartridge slot and you will see the display pictured in Fig. 6-2. Notice the square below the A in ATARI. This square is called the *cursor*. A cursor is a mark which indicates where the next character you type will appear on the screen. The ability to move the cursor to any position on the screen and change the characters being displayed is one of ATARI 800's most useful features.

A glance at the keyboard (Fig. 6-1) tells you that it closely resembles that of an ordinary typewriter. In using this keyboard, remember that each key will repeat its function rapidly if you depress it for longer than one second.

### Special Keys

Pressing either of the shift keys and holding it down while pressing another key will produce the upper case letter or the character shown on the upper half of the key, the exclamation mark or quotation mark, etc.

The control key, ctrl, functions as a second type of shift. When it is depressed in conjunction with the escape (esc) key and another key, a character from a completely new set of characters appears on the screen. These "graphic" characters can be used to produce interesting pictures, designs, and graphs either with or without the ATARI BASIC cartridge. Striking a key while holding down the ctrl key will produce the upper-left symbol on those keys having three functions.

Find the caps/lowr key on the right hand side of the keyboard

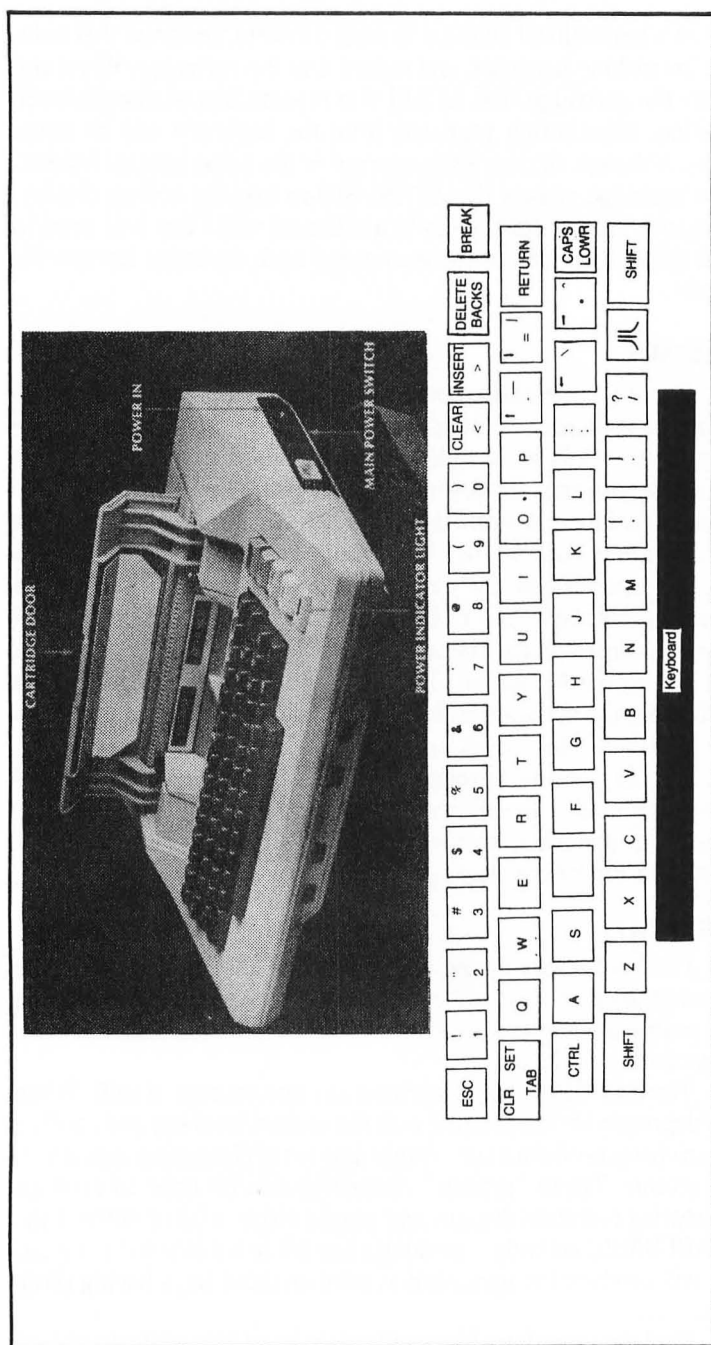


Fig. 6-1. The ATARI keyboard.



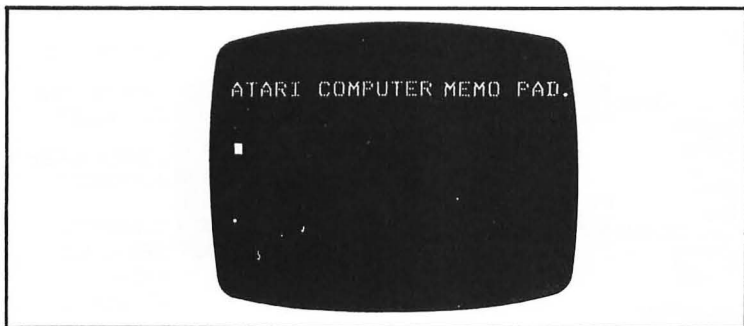


Fig. 6-2. The "memo page mode."

and press it once. You can then type lower case letters, numbers, some punctuation marks, and math symbols by pushing single keys.

You can see that the caps/lowr key locks all the alphabet keys into their alternate character displays. The non-alphabet keys—those which show two or three characters on the keytops—remain unchanged. Note that this is not the way the shift lock works on an ordinary typewriter. Try out the caps/lowr function following the chart in Fig. 6-3. You will find this feature useful when creating pictures with graphic characters and when programming in BASIC.

The return key has three functions. First, it moves the printing mechanism to the left margin and down one line. The ATARI 800 will do this automatically after 38 characters even if the return key hasn't been pushed. The largest number of characters that will fit on a single physical line across the screen is 38. However, the computer allows as many as three display lines (116 characters) to be combined into a single entity called a logical line. Logical lines will be important when using the screen editor and when programming in BASIC language.

Second, the return key can be pushed to indicate the end of a logical line for the computer. At times it will be convenient to push the return key at the end of each physical line, making it coincide with each logical line. At other times the end of a logical line will not coincide with the end of a display line.

Third, the return key activates the computer. The specific action taken depends on the software that is controlling the computer at the time the return key is pushed.

The clr-set-tab key operates much the same as the tab key on a regular typewriter. The shift and clr-set-tab keys set a tab stop at the cursor position. The ctrl and clr-set-tab keys clear the tab stop under the cursor. The clr-set-tab key by itself spaces the cursor

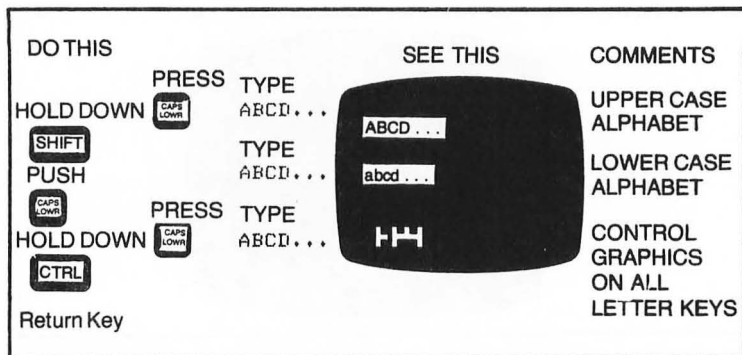


Fig. 6-3. Upper and lowercase characters.

over to the next tab stop. This key operates on logical lines so you can set tabs at any position as far as the 116th character.

The ↵ Key switches characters into inverse video. Press it again to go back to normal display.

The Break Key interrupts the computer while it is busy following instructions. Refer to the cartridge instruction sheet that came with your computer and to Fig. 6-4 for its exact function.

The Four Keys to the right of the keyboard allow you to select different starting positions within a cartridge. Each starting position is the beginning of a game or an application stored within a single cartridge. Push the system reset key to stop the computer and start from the beginning of the next game or application. Push the option key to choose among the variations possible within a game or application. After you have made your choices with the select and option keys, push the start key to begin the action. More complete instructions are provided with each cartridge.

## Edit Features

In addition to these special function keys, there are keys that allow immediate editing capabilities. These keys move the cursor and modify the display. They are used in conjunction with the shift and ctrl keys.

The following key functions are described in this section.

	CTRL CLEAR	↵
	SHIFT CLEAR	
	CTRL INSERT	
	CTRL DELETE	
CTRL ↑	SHIFT INSERT	
CTRL →		BREAK

CTRL ↓ SHIFT DELETE    ESC  
 CTRL ← DELETE

**Homing the Cursor.** Pressing the shift and clear or ctrl and clear keys simultaneously erases all characters on the screen and moves the cursor to the *home* position at the upper left corner of the screen.

**Moving the Cursor.** To move the cursor, press the ctrl key and the appropriate arrow key at the same time.

CTRL ↑: Moves cursor up one physical line without changing the program or display.

CTRL →: Moves cursor one space to the right without disturbing the program display.

CTRL	INSERT	Inserts one character space.
CTRL	DELETE	Deletes one character or space.
CTRL	1	Stops temporarily and restarts screen display without "breaking out" of the program.
CTRL	2	Rings buzzer.
CTRL	3	Indicates end-of-file.
<b>Keys Used With</b> SHIFT		
SHIFT	INSERT	Inserts one physical line.
SHIFT	DELETE	Deletes one physical line.
SHIFT	CAPS/LOWR	Returns screen display to upper-case alphabetic characters.
<b>Special Function Keys</b>		
BREAK		Stops program execution or program list, prints a READY on the screen, and displays cursor.
ESC		Allows commands normally used in direct mode to be placed in deferred mode; e.g., In direct mode, CTRL CLEAR clears the screen display. To clear the screen in deferred mode, type the following after the program line number. Press ESC then press CTRL and CLEAR together.
PRINT " ESC CTRL CLEAR "		

Fig. 6-4. Keys used with the control keys.

**CTRL↓**: Moves cursor down one physical line without changing the program or display.

**CTRL←**: Moves cursor one space to the left without disturbing the program or display.

Like the other keys on the ATARI keyboard, holding the cursor control keys for more than ½ second causes the keys to repeat. When the cursor is placed on top of a letter, that letter is shown in "inverse video" on the screen. When the cursor is moved away from the letter, the letter returns to its normal state. If the cursor is placed on top of a character and then another key is pushed, the new character will replace the original one.

**Line Insert Function:** Press the shift and insert keys simultaneously to create a space for a new line. The logical line that contained the cursor, and all lines below, it will be moved down one line. Be aware that any information on the bottom line of the screen will be lost.

**Character Insert Function:** Press the control and insert keys simultaneously to a space for a new character. The character under the cursor will be moved to the right. The rest of the line will also shift to the right. The cursor remains on the space which is now available for a new character.

**Character Erase Function:** Pressing the delete back s key erases each character as the cursor moves back one space at a time. The total length of the line remains unchanged.

**Line Delete Function:** Pressing the shift and delete back s keys simultaneously removes one whole logical line. If there are lines below the one that was deleted, they will all move up one line leaving a new blank line at the bottom of the screen.

**Character Delete Function:** Pressing the ctrl and delete back s keys erases the character under the cursor by moving all the characters to the right of the cursor one space to the left. The lines become shorter.

**The Esc (escape) Key** disables the cursor control movements and prints a graphic character instead. For example, press the esc key; then hold down the ctrl key while pressing the delete back s key. Instead of the text moving to the left, you will see a special graphics character displayed. This function will prove very useful when you begin programming in BASIC. The characters produced in this manner are shown in Fig. 6-5.

**The ⌘ Key** switches characters into inverse video. Press it again to go back to normal display.

**The Break Key** interrupts the computer while it is busy

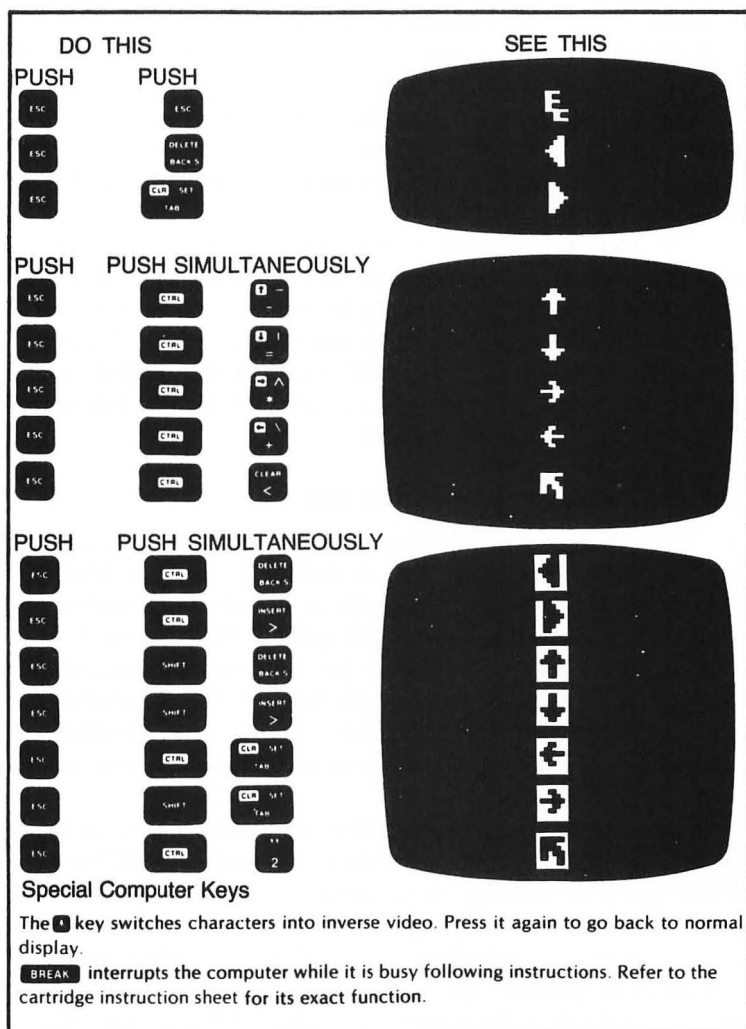


Fig. 6-5. Keys producing graphics.

following instructions. Refer to the cartridge instruction sheet that came with your computer and to Fig. 6-5 for its exact function.

## PROGRAM RECORDER

The ATARI program recorder is used with a cassette to hold blocks of software too large to be maintained in cartridge form. Programs, recorded on magnetic tape, are copied by the computer from the tape into RAM memory. You can run or modify these

programs by entering instructions through the keyboard. You can also type your own programs into memory, and then store them on tape for later use or modification.

The ATARI program recorder resembles an ordinary audio cassette tape recorder. Its playback and recording levels have been permanently set at the correct volume for use with the ATARI Computer.

The instructions with each program cassette type on the keyboard to have the computer begin to load the tape. After the program is completely loaded into the computer, the tape will stop automatically. Press the stop-eject button on your recorder to turn off the motor. Now type a command or press the computer system key labeled "start"—whichever is specified on the instruction sheet—to begin using the program.

## **LEARNING TO PROGRAM THE ATARI 800**

With the large array of preprogrammed cartridges available for the ATARI 800, one might find it difficult to imagine why it is necessary to learn how to program the computer. Once you become involved in computer use, you will quickly find that you'll have numerous uses for the computer for which no preprogrammed cartridges are available. You can, of course, hire a programmer to work the problem out for you, but this can run into the money and cost you a pretty penny. So, it's best if you learn to work out the programs for yourself.

Even if you have no practical use for computer programming, you may find that learning to write programs in BASIC for your ATARI 800 is an exciting and valuable experience. Programming sharpens your skill in thinking, in analyzing problems, and in devising step-by-step solutions. It deepens your understanding of computers in general, and no one can deny that computers are becoming a major force in modern society. A knowledge of programming makes you a more informed consumer and citizen who no longer accepts "It was the computer's fault" as an excuse for bad management.

But perhaps the most important reason to learn to program is that it is fun. Start by instructing the computer to draw pictures and to print verbal messages on the display screen. Soon you will be choosing more and more complex tasks for your ATARI 800 and will be enjoying the challenge of designing programs which allow the computer to do your bidding.

As mentioned in an earlier chapter, good computer programs

are usually created in three stages—design, coding and debugging. During design you choose a task for the computer and analyze it into its component parts. During coding you translate these parts from their English or mathematical form into a computer language, in this case ATARI BASIC. Next you type your coded program into the ATARI 800 computer. As each line is typed, ATARI BASIC will check it and report any mistakes in coding. After you have corrected these mistakes in coding, you can try to run your program. In other words, you can direct the computer to follow the set of instructions you have given it. Often you find you have made an error somewhere. The ATARI 800 may succeed in running your program, but the result is not exactly what you wanted. Perhaps you have made an error in design and need to go back and plan your program more carefully. At other times the computer will tell you that it can't follow your instructions as given because they contain logical or grammatical errors. You have made an error in coding or typing. At the third stage, debugging, you find and correct all of your remaining errors. You continue to run and debug your program until the ATARI 800, under the control of your program, produces the results you desire.

Note that no computer actually “solves problems” or “answers questions” by itself. Using your design, the computer performs the instructions you have given it. It mechanically produces a “solution” to a problem whenever you run the program.

As mentioned in Chapter 5, BASIC (Beginners All-Purpose Symbolic Instruction Code) was invented so that people could learn to write programs quickly and easily. To you, the user, the BASIC language is a set of rules stated in English which tells you how to give the computer the instructions it needs to do your bidding. To the computer, BASIC is the same set of rules written in machine language. The rules enable it to translate your BASIC instructions into action. We call this machine language program that “translates” BASIC into machine language the BASIC language interpreter. It is contained in the ATARI BASIC Language Cartridge. The program which you write is called the BASIC source program. You enter your source program into the ATARI's random access memory by typing it on the keyboard while the BASIC cartridge is in the cartridge slot. The ATARI uses the operating system programs (in the operating system ROM), the BASIC language interpreter (in the cartridge), and your BASIC source program (typed on the keyboard or loaded from cassette and stored in RAM) in order to follow the instruction you have written in your program.

## WRITING BASIC SOURCE PROGRAMS

This section will give you a brief introduction to actually programming the ATARI 800 computer—in ATARI BASIC. You were given a brief introduction to fundamental BASIC programming in an earlier chapter, but here you will actually start to press keys on the keyboard and see the results on the screen. Follow the instructions exactly, but don't expect to understand all of them at this time. You'll eventually have a good understanding of programming, but it will have to come with time and experience. If you are a programmer experienced in using other forms of BASIC, the following paragraphs will give you insights into ATARI BASIC specifically. Remember, however, there is a great deal more to learn than is presented in this chapter.

First of all, let's explore the use of nine BASIC commands: run, break, return, dim, print, input, list, if . . . then, and goto. Start out by inserting the BASIC cartridge and powering up. In the upper left hand corner of the screen, you'll see the word ready. This is the BASIC prompt. It is a message from BASIC to you telling you that the computer is waiting for a command. You'll also see a small square mark directly beneath the word ready. This mark is the *cursor*. It tells you where the next character will be printed.

Since ATARI BASIC language can only read upper case letters, press the shift and caps/lowr keys. This locks all letters into upper case. Now type an English sentence

COMPUTER, ARE YOU AWAKE?

and push return. The screen will now appear as shown in Fig. 6-6.

Yes, ATARI 800 is awake. It is sending you a prerecorded message saying that you have typed in, or entered, a line of characters which is not in its list of correct BASIC language codes. You may find out more about your mistakes in the chapter on Error Messages in ATARI BASIC in your ATARI owners' manual. Before you look at this chapter, however, try entering the line again with these changes. Type the line exactly as shown, including the word print and the quotation marks:

PRINT "ATARI 800, ARE YOU AWAKE?"

Be meticulous when you type. The computer cannot guess what you meant to include nor can it ignore extra characters. The computer will display each character on the screen as you type. The line above



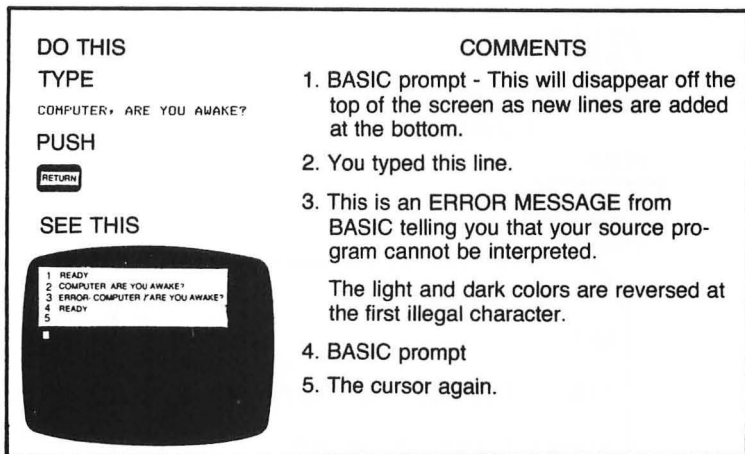


Fig. 6-6. The error message.

is correct BASIC code. Now push the return key and the new screen should appear as shown in Fig. 6-7.

If your screen display doesn't look like the one in Fig. 6-7, check the display to make sure you have copied every character exactly. Be sure to start your line with the cursor at the left margin. When you have found your mistake push the return key and try again.

Play around with the print command for a few minutes. You may put any characters you like inside the quotation marks including all the graphic and control characters described earlier in this chapter. Experiment with the cursor controls and with logical lines of up to 116 characters. What happens if you forget to put in the quotation marks? Try it and see.

Occasionally you may "lose control" of your computer by inadvertently commanding it to do something you didn't anticipate. If this happens, press the break key. If you have been using a program the computer should display

STOPPED AT LINE 10 (or some other number)

If the break key doesn't work, press the system reset key. The initial (blank) screen will be displayed.

You can also use the print command to turn your ATARI 800 into a calculator. Type the following on the keyboard.

PRINT 45782 + 11111

DO THIS  
TYPE  
PRINT "ATARI 800, ARE  
YOU AWAKE?"

PUSH

**RETURN**

SEE THIS

1. READY
2. PRINT "ATARI 800, ARE YOU AWAKE?"
3. ATARI 800, ARE YOU AWAKE

Fig. 6-7. Printing in the direct mode.

and push return. The display should appear as shown in Fig. 6-8.

The ATARI 800 evaluates expressions in the same way you were taught to do it in elementary school. Everything inside parentheses ( ) will be done first. Then exponentiation ( ^ ) will be done. Exponentiation involves multiplying a number by itself a

DO THIS  
TYPE  
PRINT 45782 + 11111  
PUSH

**RETURN**

SEE THIS

1. READY
2. PRINT 45782 + 11111
3. 56893
4. READY

Fig. 6-8. Using the computer as a calculator.

specified number of times. For example,  $5^3$  means 5 times 5 and is equal to 125. You will often see this expression written  $5^3$ . It is read "five raised to the third power". Type the exponentiation symbol on the computer by pressing the shift key and the key with the symbol  $\wedge$  on it (The  $\wedge$  symbol will be found in the upper right-hand corner of a key near the right side of the keyboard.)

All multiplication (\*) and division / is done next. Addition + and subtraction - are performed last. If you have forgotten that the order in which calculations are done is important, try a few problems. You will find that  $(4+6)/5 = 2$ , while  $4+6/5 = 5.2$ , and  $4/5+6 = 6.8$ .

The ATARI 800 will print any whole number value in the range -999999999 to 999999999. It will express fractions and numbers outside this range in scientific notation as a decimal number between one and ten times a power of ten. For example, if the value of your expression is 12,345,678,909 the ATARI 800 will write it as  $1.23456789\text{E} + 10$ .  $\text{E} + 10$  means you must move the decimal point ten places to the right to obtain the original number. You read this number as 1.23456789 times ten to the tenth power (10,000,000,000).

You can enter any expression and the ATARI 800 will do the arithmetic. Remember that the BASIC language does not tell you the correct expression to use to solve number problems. You must figure out how to calculate interest payments, sales commissions, expected time of arrival, or whatever you want to know. BASIC does have a large repertoire of arithmetic, algebraic, and trigonometric functions available, but you, the programmer, must tell the computer how to combine them to produce meaningful results. You will find a complete discussion of BASIC language arithmetic in a later chapter.

Thus far, we have been using ATARI BASIC language in the direct mode; that is, the ATARI 800 follows your command as soon as you give it and then forgets about it. In the program mode, you may give the computer as many as about two hundred lines of instructions. Each line is then stored until the run command is entered. Then the computer follows one instruction after another until the whole task is completed.

Although most BASIC commands can be given in direct mode, only a few of them are useful to the beginner. The run command is almost always used in direct mode. It is used to start a program at the beginning. It is typed as a three letter sequence in upper case

only. No action occurs until you press the return key. This key signals the computer to read what you have typed and to execute the entire series of instructions. Pressing the break key stops the program from running and displays the cursor. Break and return are the only single key commands in BASIC. All others are typed as a series of letters followed by the return key.

If you wish to enter a program into the ATARI 800 using the program mode, start each line at the left margin of the screen with a line number. You may number your lines with any whole numbers between one and 32767. You may enter program lines in any order. However, the computer will rearrange them and execute them starting with the line with the smallest number and proceeding to the next higher numbered line until all instructions have been followed. Most BASIC language programmers number their lines by tens so that they have nine line numbers available in case they wish to add new lines between the original ones.

To get started, power down and up once again. This will clear out your RAM memory. Then print:

10 PRINT "WIZARD, ARE YOU AWAKE?"

Continue by pushing return and then typing run; then press return again. The display should appear as shown in Fig. 6-9. You have just entered and run a BASIC program!

To edit any line of BASIC code, use the display control keys. Position the cursor over the character you wish to change and make the corrections within a logical line. When the line is correct, press the return key. The ATARI 800 will put the corrected line in place of

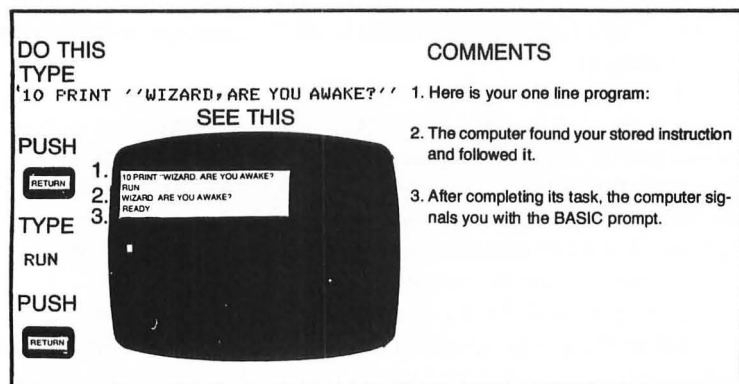


Fig. 6-9. Printing in the deferred mode.

the original one in RAM. To remove a whole line, type its line number and press the return key. For practice, replace the word "Wizard" in line 10 with your own name. Be sure to make your changes in the numbered line 10, not the line displayed after "run."

List is the command that makes the computer display your BASIC source program. Any time you wish to see the instructions your computer is following, press break, type list, and press the return key. Note that you may use the break key to interrupt the computer even if it is running a program or calculating. All other commands must be given only after the cursor is displayed. Commands must be typed in upper case or you will get an error message. Now add the following lines to your program. Remember to press the return key before beginning a new line that starts with a line number. Be sure not to leave out the colons.

```
5 DIMA$(10)
20 Input A$
30 IF A$ = "YES" THEN PRINT "YOUR PROG
   RAMMING CAREER HAS BEGUN.":GOTO 60
40 IF A$ = "NO" THEN PRINT "TECHNOLO
   GY MAY PASS YOU BY . . .":GOTO 10
60 PRINT "THIS PROGRAM HAS ENDED."
```

You should know that BASIC ignores blank spaces unless they are inside quotation marks so you can leave them out when you need to save space. However, do put them in when you can so your listings will be easier for people to read.

Run this program and answer the question "Are you awake?" several times. Type your answer after the ? and then push the return key. Run the program several times more. Try responding Yes; YES; yes; No; and maybe. What happens when your answer is neither "YES" nor "NO"? The program should print the message in line 60 and then end.

To understand what the computer is doing, examine the program line by line. Line 5 sets aside a place in the computer's memory for your answer to the question in line 10. Line 10 has the command print "something." This something that you put between the quotation marks is called a string constant: string because it is a string of letters, numbers, and/or graphic characters, and constant because it will remain the same every time you run this program.

Line 20 has the command "input something". This means:

"Computer, wait until something is typed on the keyboard. Show whatever is typed on the screen and store it in a space in RAM memory labeled A\$. When the return key is pressed, go on to the instruction with the next higher line number." A\$ (read A string) is called a string variable because it is a string of letters, numbers, and/or graphic characters which will be different every time you run this program. Each string variable within a single program must have a different label. Many labels are available in BASIC, but for now, limit yourself to A\$ through Z\$. Line 5 gave us space to store ten characters in A\$. If we expected a long answer we could save more memory for A\$ by using a large number within the parentheses.

Line 30 has `if . . . then print "something": GOTO 60`. When the condition following `if` is met, that is, when the computer finds the letters Y-E-S stored in the place in memory labeled A\$, the instructions in the rest of the line are followed. It prints "Your Programming career has just begun." The computer encounters the colon (:) next. This tells it that another command is coming. The GOTO command says "skip down to line 60 and ignore the lines in between."

When the condition is not met, that is, when the computer finds that anything besides Y-E-S has been entered at line 20, the rest of the line is ignored and the computer goes on to the next numbered line, in this case, line 40.

Line 40 operates the same way as line 30 does. If anything but N-O has been entered at line 20, its condition isn't met, and line 40 doesn't print anything. The computer just goes on to the next numbered line which is line 60. Line 60 prints its message, and since there are no more lines, the program ends and ready is displayed. If N-O has been entered at line 20, all of the line 40 would be executed. The message in line 40 would be printed and program control would jump back to line 10. The program would begin over again.

Suppose the programmer wants to demand an answer of "YES" or "NO" before the program will end. The computer must be programmed to print a message whenever something other than "YES" or "NO" has been entered. After this message, the question should be repeated on the display screen. Try to figure out what line to add to make the computer do this. There are many ways, but one solution would be:

```
50 PRINT "YOU MUST ANSWER 'YES' OR 'NO'
   TO GO ON.":GOTO 10
```

Enter this line. List and run this revised program several times. To anticipate what the display will look like, pretend you are the computer and follow the program. Remember that computers are machines which blindly stick to their programs no matter how nonsensical the results may be. This is why the design stage of programming is so important. It is up to you, the programmer, to plan each display the user will see, to anticipate what the user will type in response to that display, and to tell the computer what to do with every response.

You can use the previous format to code an unlimited number of dialogue-type programs. You will need to change the string constants and occasionally rearrange the GOTO's. Here is another example . . . a type of quiz.

```
10 DIMN$(100)
20 DIMA$(20)
30 DIMP$(20)
110 PRINT "THIS IS THE OCEANS PROGRAM."
120 PRINT "HELLO, WHAT IS YOUR NAME";
130 INPUT N$
140 PRINT "OK, ";N$;
150 PRINT ".NAME AN OCEAN THAT BEGINS WITH P."
160 INPUT P$
170 IF P$ = "PACIFIC" THEN PRINT "PERFECT.":GOTO
    190
180 PRINT "THAT ISN'T THE ONE WE ARE LOOKING FOR.
    TRY AGAIN."; GOTO 150
190 PRINT "NOW NAME ONE THAT BEGINS WITH A."
200 INPUT A$
210 IF A$ = "ATLANTIC" THEN PRINT "YOU'RE TOO
    GOOD, "N$;" . NOW GIVE SOMEONE ELSE A
    TURN.":GOTO 120
220 PRINT "TRY AGAIN. REMEMBER SPELLING
    COUNTS!":GOTO 190
```

The display on the screen will appear as shown in Fig. 6-6 for this particular program. See if you can follow the program and the answers on the screen. Does it make sense?

## EDITING YOUR PROGRAM

To perfect your program, you will want to use the keys described under "Edit Features" earlier in this chapter. When you are using these keys, remember that the keyboard and display are

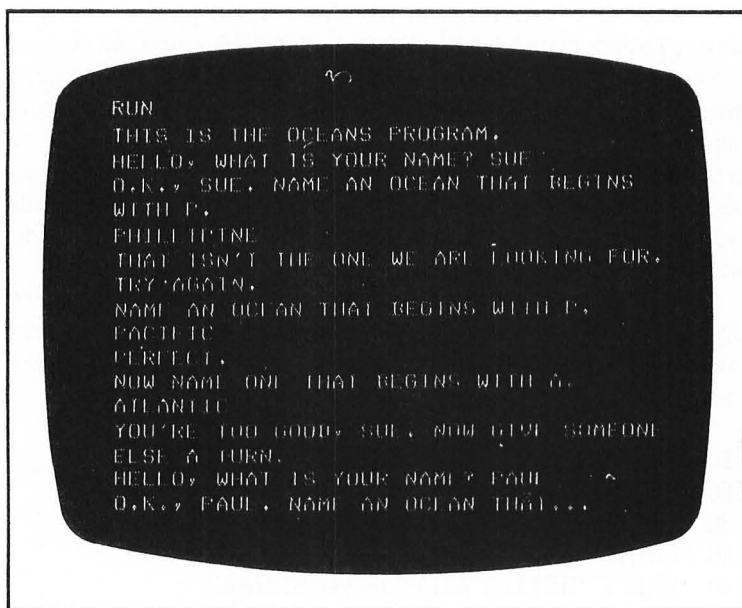


Fig. 6-10. A question and answer program.

logically combined for a mode of operation known as screen-editing. Each time a change is completed on the screen, the return key must be pressed. Otherwise, the change is not made to the program in RAM.

Example: 10 REM PRESS RETURN AFTER LINE EDIT  
20 PRINT :PRINT  
30 PRINT "THIS IS LINE 1 ON THE SCREEN."

To delete line 20 from the above program, type the line number and press the return key. Merely deleting the line from the screen display does not delete it from the program.

## TRANSFERRING PROGRAMS TO AND FROM CASSETTE TAPES

Now that you have a BASIC program entered into the computer, and you know that it runs, you may want to save it on tape. Once you have a copy of the program on tape, you will never have to type that particular program again. When you want to change your program, you can load your first version from the tape; edit the old lines and add new ones; then save the new version. The copy in the computer disappears each time you turn off your computer or load



another program, but the copy on the tape will be available until you choose to erase it.

Set up your program recorder according to the instructions provided by the manufacturer. See Fig. 6-11. If you already have a program in the computer, be sure not to shut the power off at any time. If you have not yet written a BASIC Source Program, write one.

To transfer a program to tape:

1. Insert a blank cassette tape into the program recorder with the recording surface toward you and the label so that you can read it.
2. Push the rewind button and wait until the tape stops.
3. Push the tape-counter reset button until it reads 0000.
4. Push the stop-eject button once.
5. Type `CSAVE` on the computer keyboard. Then press the return key. You should hear two beeps.
6. Push the record and play buttons simultaneously. Now push the return key on again. You will hear a series of tones indicating that the recorder is now being controlled by the computer. The recorder will erase the beginning of the tape surface for eighteen seconds; write the introductory header needed by the computer; copy your program onto the tape; then stop.

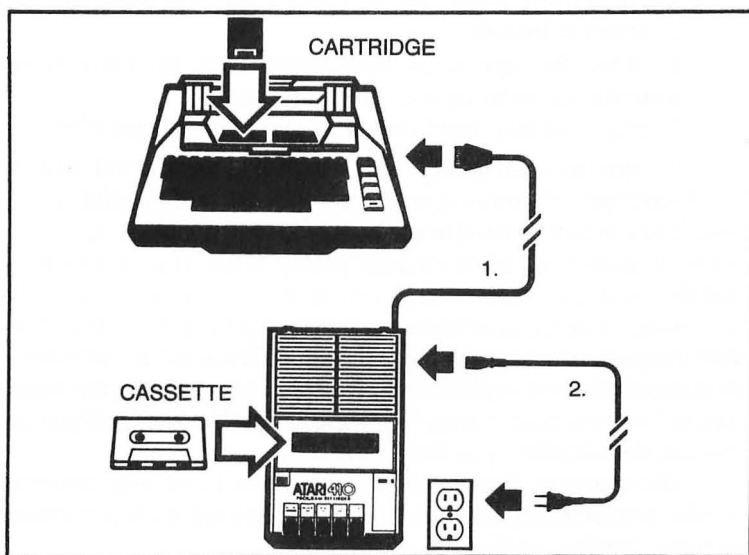


Fig. 6-11. Connecting the cassette recorder.

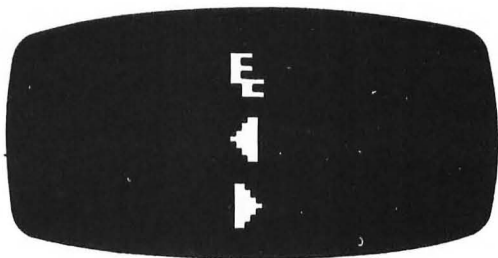
7. It is always good programming technique to create a backup copy of each program you wish to save. ATARI recommends that you store one program on each cassette and keep a backup on a separate cassette—just in case!
8. If you are planning to store more than one copy of the program on the tape, you may make your backup by repeating steps 5 and 6. Be sure to write down the starting tape-counter number for your backups as well as for your original programs.
9. Push the stop button on the program recorder.
10. Finally, write the name of your new program and its tape-counter number on the cassette label and on a page which you keep in the binder with your manuals.

To load a program from tape to the computer.

1. Insert BASIC source tape into the program recorder.
2. Rewind tape to the beginning and press the stop-eject button.
3. Press the tape-counter reset button until it reads 000. If you have put more than one program on a single tape, advance the tape to the number where your program starts.
4. Type CLOAD on the computer keyboard. Then press the return key. You will hear one beep.
5. Push the play button on the program recorder. Then push the return key again. You will hear a series of tones as your program is loaded.
6. When the tape stops, your program has been transferred from the cassette tape to the computer.
7. Push the stop-eject button on the program recorder.

We have covered quite a lot of material in this chapter so don't be discouraged if you're somewhat confused at this point. Many people are inexperienced in the kind of logical thinking required to write programs, so allow yourself plenty of time to become comfortable with the step-by-step nature of the computer, and the sometimes infuriating attention you must pay to detail. Remember that inexperience does not mean inability. There are a tremendous number of new and sophisticated concepts presented in this book. You will want to read it many times and keep it handy so that you can look up details when you forget them.

The chapters that follow will give you more experience in BASIC programming on the ATARI 800. They will include methods of using graphics and sound.



### Optional Peripherals and Software

The ATARI 800 can be customized by adding components that are appropriate for the tasks you have at hand. By choosing a particular combination of preprogrammed software, controllers, memory modules, and peripheral components, you can completely change the character and capacity of the machine.

#### 8K COMPUTER SYSTEMS

The ATARI 800 basic system contains 8K of random-access memory and the ATARI 410 program recorder. This system configuration will accommodate all of the preprogrammed cartridges and cassettes which make up ATARI'S 8K library. Some of the software in the 8K library will require the use of hand held controllers which are available from ATARI retailers.

In addition to the preprogrammed 8K library, the BASIC language cartridge may be used in conjunction with the 410 program recorder and any high-quality audio cassette to create and store an unlimited number of programs that you may want to write yourself. The ATARI BASIC language offers you the opportunity to use sophisticated graphic displays, four-voice sound, and input from the controllers of your choice in programs limited only by your imagination.

#### THE PRINTER

The ATARI 820 printer may be added to any ATARI 800

system. Some programs in the 8K library will offer the option of printing program results as well as displaying them on the screen. The 820 is a high-resolution, dot-matrix, impact printer which uses standard 4-inch roll paper and will output up to forty characters per second. Your printer should be connected in a "daisy chain" series between the serial jack labeled PERIPHERAL on the side panel of the ATARI console and the program recorder. See your manual for details.

## **16K COMPUTER SYSTEMS**

The basic ATARI 800 system comes with one 8K random-access memory module installed. This module contains approximately 8,000 storage cells to use. The capacity of the computer may be expanded to enable you to use longer programs by inserting additional RAM memory modules into one or both of the empty sockets in the memory bank. These modules may be purchased in 8K and 16K versions. The 8K version is ATARI part number CX852. The 16K version is part number CX853.

These additional memory modules may be inserted in any combination (see Fig. 7-1), except that the 10K ROM must remain in socket 0. Also, socket 1 must be filled first; then socket 2, and finally socket 3. Note that the ninth option, 48K of memory, disables the cartridge slots and should only be used after gaining some experience operating the computer.

The memory bank is located under the ribbed top cover of the console. You may install modules yourself by opening the cover and arranging the modules in one of several configurations as shown in Fig. 7-1. To install these modules, first, turn the power off using the main power switch, and open the cartridge door. Next rotate the two black clamps outward as far as they will go. Lift up the front edge of the cover slightly and slide the entire hinged door and ribbed top cover toward you. Inside the memory bank you will see four module sockets, two of which are already occupied by the operating system 10K ROM and one 8K RAM module. The operating system 10K ROM must remain in the front socket. The other three sockets are available for expanding the computer's RAM.

When inserting modules, press firmly on both sides and push straight down. The ATARI 800 may then be reassembled by slipping the back of the ribbed cover into place first. The two metal tabs on the underside of the cover fit into two slots at the back edge of the console. Shut the front edge of the cover and replace the blank

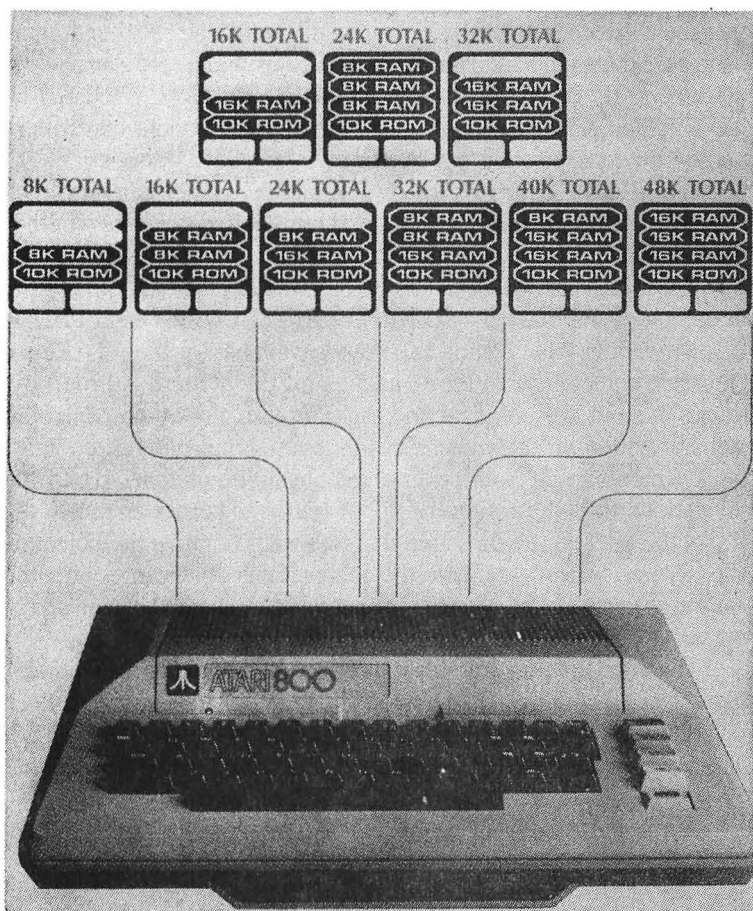


Fig. 7-1. The ATARI console.

plastic clamps by rotating them forward. Close the cartridge slot door and power up again.

## CONTROLLERS

There are several different controllers used with the ATARI 800. Each of them can be attached directly to the computer or to external mechanical devices so that outside events can be fed directly to the computer for processing and control purposes.

**Paddle:** A paddle consists of a knob that can be turned to send a number between 1 and 228 to the computer. The number increases as the knob is rotated counterclockwise. This number can

be used with other functions or commands to “cause” further actions such as changes in sound or graphics. For example, you can use the statement `IF PADDLE(3) = 14 THEN PRINT “PADDLE ACTIVE.”` The ATARI console can control eight paddle controllers numbered zero through seven from left to right.

**Paddle Trigger (PTRIG):** The paddle trigger is a button on the paddle. You can use the PTRIG statement to determine whether the button is pressed (returns a value of 0) or not pressed (returns a value of 1) on a particular paddle (0-7). For example, you can use the statement `IF PTRIG (3) = 0 THEN PRINT “BOMBS DROPPED!”`

**Joystick:** Many game cartridges available for the ATARI use “joystick controllers to move images on the screen. All four joysticks in a set are identical and can plug into any of the controller jacks provided on the console. Each joystick has one button and a stick which can be pointed in eight possible positions. Hold the joystick with the button in the upper left-hand corner and push the top of the stick in the direction that you want to move the object on the screen. Always consult the instructions that came with the cartridge to determine whether or not joystick should be used and, if so, what each position means.

If you wish to use the joystick in programs you write yourself, use the stick command just as you used the paddle command. Each joystick position will return a different number as shown in Fig. 7-2. The joystick controllers are numbered from 0-3 from left to right.

First controller	STICK(0)
Second controller	STICK(1)
Third controller	STICK(2)
Fourth controller	STICK(3)

**Stick Trigger (STRIG):** The button on the joystick is the stick trigger. You can use the STRIG statement for the stick trigger in the same way you used the PTRIG statement for the paddle trigger. You can also use the STRIG statement to evaluate the status of a key on the keyboard that you have designated as a controller.

## THE ATARI 810 DISK DRIVE

The ATARI 810 disk drive provides fast, reliable data storage and has the capabilities to:

1. Store programs on diskette.

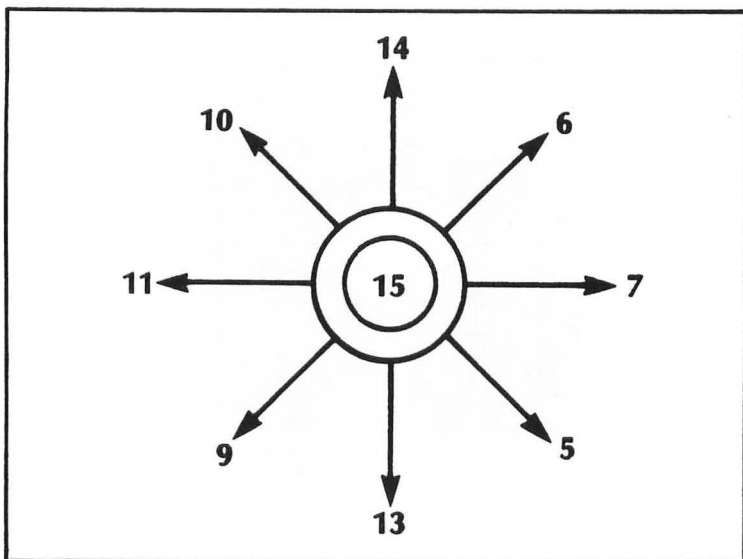


Fig. 7-2. Joystick controller movement.

2. Retrieve programs from diskette.
3. Create and add to data files needed by programs.
4. Make copies of disk files.
5. Erase old files from a diskette.
6. Load and save binary files.
7. Move files to and from memory, screen, diskette, printer, and any new peripherals that ATARI may introduce.

Each disk drive contains a microprocessor with its own software in addition to the hardware necessary for the magnetic head to move back and forth "across" a floppy diskette. The ATARI 800 with 16K of RAM can accommodate up to four ATARI 810 disk drives, and each will operate independently of the others.

ATARI diskettes are thin, circular mylar sheets covered with an oxide similar to that used on recording tape. They are protected by a special, black jacket which keeps them from being bent, scratched, or contaminated. See Fig. 7-3.

Before writing on a blank diskette, the surface must be "organized" so that the disk operating system (see the next section) will know where the information is located. This is accomplished by *formatting* a diskette into *tracks* and *sectors*. The diskette is logically divided into 40 tracks with 18 sectors per track. Each of these

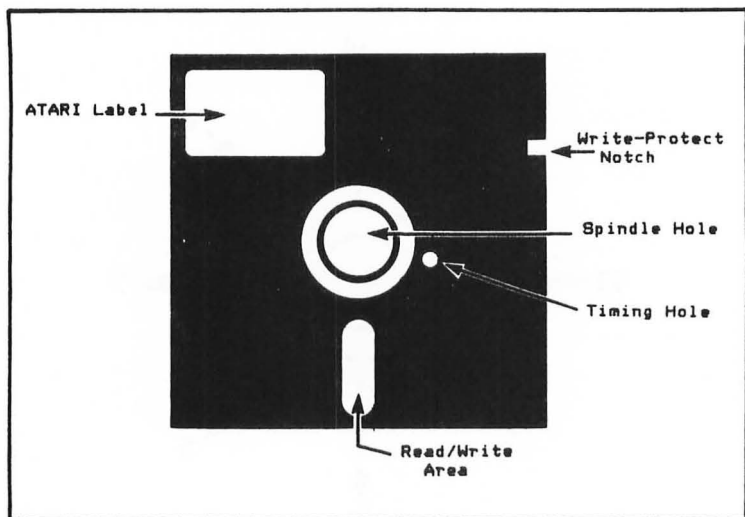


Fig. 7-3. An ATARI diskette.

*single-density* diskettes has a total of 720 sectors on which information may be written.

Eleven of the sectors are allocated by the DOS for “special duty” so they are not available to you.

- 1 sector is used for the *boot*.
- 8 sectors are used for the *directory*.
- 1 sector is used for the volume table of contents.
- 1 sector (number 720) is not addressable and so is unused.
- 11 TOTAL

Each of the remaining 709 sectors can store 128 bytes of information, 3 bytes of which are used for overhead. That gives a total byte capacity per single-density diskette of 88,625 bytes!

## DISK OPERATING SYSTEM

DOS (pronounced doss) is an acronym for disk operating system. It is an extension of the ATARI operating system (OS) that allows interfacing with a disk drive so that information can be passed between the diskette and the computer memory (RAM).

The operating system contains two main parts: a disk utility package (DUP) and a file management subsystem. When the system is energized, it executes a *bootstrap* loader (self loader) that brings a special file called DOS SYS into RAM and begins executing the



initial code in that file. DOS SYS contains both the file management subsystem and the disk utility package.

When DOS is loaded into the computer RAM, it occupies a special area in RAM that does not interfere with the memory locations allocated for BASIC or assembly language programs. The memory map in Fig. 7-4 shows the RAM locations occupied by the DOS. After the disk drive system has been booted and the DOS SYS file is loaded, the ATARI operating system turns control of the system to the cartridge. If no cartridge has been inserted, OS gives control of the system directly to the disk utility package (DUP).

The executive program in the DUP allows you to display the DOS menu which lists the commands available to you. For example, DELETE FILESPEC is menu selection D. It then executes the selected utility. The DUP program also allows easy access to the file management subsystem (FMS) without your having to understand the logical structure of FMS.

ADDRESS		CONTENTS
DECIMAL	HEXADECIMAL	
65535	FFFF	OPERATING SYSTEM ROM
	E000	
10879	2A7F	DISK OPERATING SYSTEM (2A7F-700) DISK I/O BUFFERS (current DOS)
9856	2680	
9885	267F	DISK OPERATING SYSTEM RAM (current DOS)
4864	1300	
4863	12FF	FILE MANAGEMENT SYSTEM RAM (current DOS)
1792	700	
1791	6FF	FREE RAM
1536	600	
1151	47F	OPERATING SYSTEM RAM (47F-200) CASSETTE BUFFER
1021	3FD	
999	3E7	PRINTER BUFFER
960	3C0	
511	1FF	HARDWARE STACK
256	100	
255	FF	PAGE ZERO FLOATING POINT (used by BASIC)
212	D4	
211	D3	BASIC or CARTRIDGE PROGRAM
210	D2	
209	D1	FREE BASIC RAM
208	D0	
207	CF	FREE BASIC and ASSEMBLER RAM
203	C9	
202	CA	FREE ASSEMBLER RAM
176	B0	
128	B0	ASSEMBLER ZERO PAGE
		BASIC ZERO PAGE

Fig. 7-4. Memory map (partial).

## File Management Subsystem

This system provides a simple way of storing data on a diskette by putting a logical structure on top of the formatted diskette. Because of the file manager program, it is not necessary to keep track the number of the sectors a program occupies, which sectors they are, or how to find a particular file. FMS relieves the user of all that responsibility.

Operations on a file, such as open, close, put, and get are also controlled by the FMS. In addition, it defines the subfunctions displayed on the DOS menu and are accessed by the DUP. The subfunctions that are not defined by FMS are binary load, binary save, run at an address, run cartridge, copy file, duplicate file and duplicate disk.

The FMS organizes files using filenames you have assigned to your files.

The disk directory contains a list of all the files on a diskette. On demand, it displays the filename, the extender, and the number of sectors allocated to that file. It will either display a partial list or a complete list depending on the parameters entered. *Wild cards* can be used in the parameters.

The ATARI DOS recognizes two “wild cards” which can be substituted for characters in a filename. They are represented by the special characters ? and \*. The first, ?, is used in place of any single valid character. If there are 25 files on a diskette and you want to display a list of those five-letter file names that begin with PROB and have the *extender* (ending) .BAS, you would type PROB?.BAS. This wild card can only be substituted for a single character. To substitute for a variable number of characters, there is another, more flexible wild card.

The \* stands for any valid combination of characters in a filename or an extender. The following examples illustrate the use of the \*.

*.BAS	will list all the program files on a diskette in drive 1 that end in .BAS.
-------	--

D2:*. *	will list all the program files on the diskette in drive 2.
---------	---

PRO*.BAS	will list all the program files on the diskette in drive 1 that begin with PRO and have .BAS as the extender.
----------	---

A program may be loaded from the diskette using a wild card in the file name if there is no more than one file to which it is applicable.

The ATARI 810 also has the capabilities to copy files. For example, if you desire to copy a file from one diskette in drive 1 to a second diskette in drive 2, this can be easily done. You can also create a backup copy of a file on the same diskette by using the same filename with a different extender, or even by using a completely different filename.

## **BASIC COMMANDS USED WITH DISK OPERATIONS**

BASIC commands used to store and retrieve files are load, save, list, and enter. Other BASIC commands used with disk operations are note, point, and DOS.

### **LOAD(LO.)**

Format: LOAD filespec

Example: LOAD "D1:JANINE.BRY"

This command causes the computer system to load the file from the disk drive specified into the RAM. (Note here that it loads the *tokenized* version of the program. The tokenized version is shorter than the untokenized version in that, when recorded, this version contains shorter interrecord gaps. However, when a tokenized version is loaded, it also loads the program's symbol table. If the program is altered or deletions are made, the symbol table is NOT changed. This means that all variables which are defined in the original program still exist in the symbol table. Therefore, it is recommended that load and save be used only when saving a program is its final form.) When loading a file from drive 1, it is not necessary to specify the drive number.

The load command can also be used as a BASIC statement to "chain" programs. If you have a program that is too big to run in the available RAM, you can use the load statement as the last line of the first program. When the program encounters the load statement, it will automatically read in the next part of the program from the diskette. However, the second program must be able to stand alone without depending on any variables, and the like from the first program.

### **SAVE(S.)**

Format: SAVE "filespec"

Example: SAVE "D1:JANINE.BRY"

This command causes the computer system to save a program on a diskette using the file name designated in the command. Save is the complement of load and stores programs in tokenized form.

### **LIST(L.)**

Format: LIST "filespec"

Example: LIST "D: DEMOPR.INS"

When a disk drive is specified, the list command causes the computer system to move the program currently in RAM to a file on the diskette using the filename specified in the command. This command unlike save, saves the file in untokenized format.

### **ENTER(E.)**

Format: ENTER "filespec"

Example: ENTER "D:DEMOPR.INS"

This command causes the computer to move the file you specify from the diskette into RAM. The program is entered in untokenized form and is interpreted as the data is received. Enter, unlike load, will not destroy a RAM-resident BASIC program, but will merge the RAM-resident program and the disk file being loaded. If there are duplicate line numbers in the two programs, the line in the program being entered will replace the same line in the RAM-resident program.

### **NOTE(NO.)**

Format: NOTE #aexp,avar,avar

Example: 100 NOTE #1,X,Y

This command is used to store the current disk sector number in the first arithmetic variable, (avar) and the current byte number (within the sector) in the second avar. The numbers specify the current read or write position and thus indicate where the next byte to be read or written is located. The note command is used when writing data to a disk file. The information in the note command is written into a second file which is then used as an index into the first file.

Note: aexp indicates an arithmetical expression. The item placed here must have a numerical, not a string value.

### **POINT(P.)**

Format: POINT #aexp,avar,avar

Example: 100 POINT #2, A,B

This command is used when reading a file into RAM. The first avar specifies the sector number and the second avar specifies the

byte within that sector where the next byte will be read or written. Essentially, it moves a software-controlled pointer to the specified location in the file. This gives the user "random" access to the data stored on a disk file. The point and note commands are discussed in more detail in your DOS Manual.

## **DOS(DO.)**

Format: DOS

Example: DOS

The DOS command is used to go from BASIC to the disk operating system (DOS). If the disk operating system has not been booted into memory, the computer will go into "*Memo Pad*" mode and the user must press the system reset button to return to the direct mode. If the disk operating system has been booted, the DOS menu is displayed. To clear the DOS menu from the screen, press the system reset button. Control then returns to BASIC. Control can also be returned to BASIC by selecting B (Run Cartridge) on the DOS menu.

## **DIRECTORY OF BASIC WORDS USED WITH DISK OPERATIONS**

The following is an alphabetical directory of BASIC *reserved* words (words reserved for BASIC) used with disk operations. Note that the period (.) is mandatory after all abbreviated keywords.

RESERVED WORD:	ABBREVIATION	BRIEF SUMMARY OF BASIC STATEMENT
CLOSE	CL.	I/O statement used to close a disk file at the conclusion of I/O operations.
DOS	DO.	This command causes the menu to be displayed. The menu contains all DOS utility selections.
END		Stops program execution; closes files; turns off sounds. Program may be restarted using cont.
ENTER	E.	I/O command used to retrieve a listed program in unto-

kenized form. If a program or lines are entered when a program is resident in RAM, enter will merge the two programs.

GET	GE.	Used with disk operation to input a single byte of data into a specified variable from a specified device.
INPUT	I.	This command requests data from a specified device. The default device is E: (Screen Editor).
LIST	L.	This command outputs the untoknized version of a program to a specified device.
LOAD	LO.	I/O command used to retrieve a saved program in tokenized form from a specified device.
OPEN	O.	Opens the specified file for input or output operations.
PRINT	PR.or ?	I/O command causes output from the computer to specified output device.
PUT	PU.	Causes output of a single byte of data from the computer to the specified device.
SAVE	S.	I/O statement used to record a tokenized version of a program in a specified file on a specified device.
STATUS	ST.	Calls status routine for a specified device.

RESERVED Word:	ABBREVIATION	BRIEF SUMMARY OF BASIC STATEMENT
TRAP	T.	Directs execution to a specified line number in case of an input error, allowing the user to maintain control of program.
XIO	X.	General I/O statement used in a program to perform DOS menu selections and specified I/O commands.

### **BASIC ERROR MESSAGES USED DURING DISK OPERATION**

The BASIC error messages used by the ATARI 810 disk operating system are:

ERROR CODE NO.	ERROR CODE MESSAGE
2	Insufficient memory 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 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 32767: Value is not a positive integer or is greater than 32767.
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 already been dimensioned, or a reference has been made to an undimensioned array or string.
- 10       Argument Stack Overflow: There are too many gosubs or too large an expression.
- 11       Floating Point Overflow/Underflow Error: Attempted to divide by zero or refer to a number larger than +1E98 or smaller than -1E99.
- 12       Line Not Found: A gosub, goto, or then referenced a nonexistent 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.
- 19       Load Program Too Long: Insufficient memory remains to complete load.



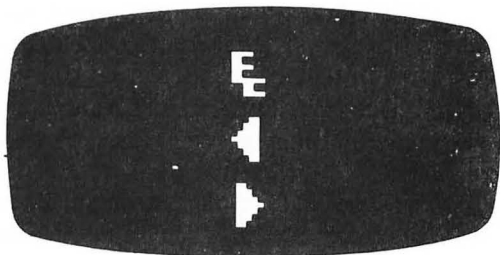
- 20 Device Number Larger than 7 or Equal to 0.
- 21 Load File Error: Attempted to load a nonload file.

The following are input/output errors that result during the use of disk drives, printers, or other accessory devices.

- 128 Break Abort: User hit the break key during I/O operation.
- 129 Input/output Control Block (IOCB) already open: Open statement within a program loop or IOCB already in use for another device or file.
- 130 Nonexistent Device Specified: Device is not turned on or not attached.
- 131 IOCB Write Only: Command to a write-only device (Printer).
- 132 Invalid Command: The command is invalid for this device.
- 133 Device or File not Open: No open specified for the device.
- 134 Bad IOCB Number: Illegal device number.
- 135 IOCB Read Only Error: Command to a read-only device.
- 136 EOF: End of File read has been reached. (NOTE: This message can occur when using cassette files.)
- 137 Truncated Record: Attempt to read a record longer than 256 characters.
- 138 Device Timeout. Device doesn't respond. Check connections between peripheral equipment and console.
- 139 Device NAK: Garbage on serial port or bad disk drive.
- 140 Serial Bus input framing error.

- 141        Cursor Out of Range for particular mode.
- 142        Serial Bus Data Frame Overrun.
- 143        Serial Bus Data Frame Checksum Error.
- 144        Device Done Error (invalid “done” byte): Attempt to write on a write-protected diskette.
- 145        Read After Write Compare Error (disk handler) or bad screen mode handler.
- 146        Function not Implemented in handler.
- 147        Insufficient RAM for operating selected graphics mode.
- 160        Drive Number 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 Number Mismatch: Links on disk are messed up.
- 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.

## Chapter 8



### I/O Operations

This chapter describes input/output (I/O) devices and how data is moved between them. The commands explained in this chapter are those that allow access to the input/output devices. The input commands are those associated with getting data into the RAM. The output commands are those associated with retrieving data from RAM.

As stated before, in order for the computer to be a useful machine, it must be able to communicate with its user. The user must be able to supply the computer with the programs and data which enable it to perform some useful operation. In addition, the computer must be able to provide an output showing the results of the computations. The input/output section of the ATARI 800 provides this communications capability between the central processing unit and the outside world.

The I/O section of the computer *buffers* (stores temporarily) and controls data transfers between the computer memory and the various peripheral devices. In the ATARI 800, the data transfers take place in the central processing unit. They are under the control of a computer program being executed by the CPU.

In small digital computers the I/O section is extremely simple, consisting generally of nothing more than some *gating* (devices that output a signal when specified input conditions are met) and simple control circuitry. In such computers, the data transfers take place through the accumulator register. To read a word in from a

peripheral device, the computer executes an input instruction and stores that word in the accumulator. Other instructions in the program then cause that word to be stored in some specified memory location and/or used in a calculation.

Output operations are performed by first loading the accumulator with the desired word. An output instruction then causes that word in the accumulator to be transferred to the peripheral device. This basic form of I/O system is simple and desirable for many types of computer operations. However, it is very slow and inefficient.

The simplest form of input/output for any digital computer is a system of binary switches and lights. A group of switches can act as a temporary storage register for a data word. The switches can be set to their off or on positions corresponding to the binary ones and zeros in the word to be used or entered. A button on the computer can be used to store that word. At the same time, output can be accomplished by placing lights on the various registers in the machine. The lights will indicate the ones and zeros stored, for example, in the accumulator register. The operator can read the numbers stored there by interpreting the binary number represented by the lights. Binary switches and lights provide a low cost, simple means of input/output operations in a computer. All digital computers have this type of I/O capability.

Although binary lights and switches are widely used, they are entirely inadequate for the bulk of most computer applications. Consider the problem involved in storing, say, a 1000 word instruction program plus data in the computer's memory by using only a set of input switches. The process would be extremely time-consuming, and the chances of making an error would be great. This large program may produce a substantial number of output results that the programmer wishes to observe. In such case, the programmer would have to observe the resulting words one at a time on the binary lights of the accumulator register. This would be time-consuming and inconvenient. As a result, more sophisticated input/output methods are required.

Most of the I/O data in a digital computer is handled by peripheral devices. Peripheral devices are machines that communicate with the computer through the input/output section. You have already seen several of these devices in Chapter 7. Peripheral devices are, in effect, a man-made interface. These peripheral devices not only speed up the input/output process tremendously,

but they also make it more convenient for the programmer and the operator. Most peripheral devices use the decimal number system and alphabet so that the operators and programmers do not have to deal with binary numbers. In the ATARI 800, the input/output keyboard simplifies the I/O process.

## INPUT/OUTPUT DEVICES

The Central Input/Output (CIO) subsystem on the ATARI 800 provides the user with a single interface to access all of the system peripheral devices in an independent manner. This means there is a single entry point and a device-independent calling sequence. Each device has a symbolic device name used to identify it; K, for example, is used for keyboard. Each device must be opened before it can be accessed, and each must be assigned to an I/O control block. From then on, the device is referred to by its IOCB number.

ATARI BASIC contains 8 blocks in RAM which give the operating system the information it needs to perform the I/O operation. This information includes the command, buffer length, buffer address, and two auxiliary control variables. ATARI BASIC sets up the IOCB's, but you must specify which IOCB to use. BASIC reserves IOCB 0 for I/O to the Screen Editor; therefore you may not request IOCB 0. The graphics statement opens IOCB 6 for input and output to the screen. IOCB 7 is used by BASIC for the LPRINT, CLOAD, and CSAVE commands. The IOCB number may also be referred to as the device number. IOCB's 1 through 5 are used in opening the other devices for input/output operations. If IOCB 7 is in use, it will prevent LPRINT or some of the other BASIC I/O statements from being performed.

The ATARI 800 uses the following, I/O devices:

**Keyboard.** (K:) Input only device. The information you enter is displayed on the screen as each key is pressed.

**Line Printer:** (P:) Output only device. The line printer prints ASCII (American Standard Code for Information Interchange) characters a line at a time. It recognizes no control characters.

**Program Recorder:** (C:) Input and Output device. The recorder is a read/write device which can be used as either one or the other but never as both simultaneously. The cassette has two tracks for sound and program recording purposes. The ATARI system cannot put sound on the audio track but the sound on the track may be played back through the TV speaker.

**Disk Drives:** (D1:,D2:,D3:,D4:) Input and Output devices. If

16K or RAM is installed, the ATARI can use from one to four disk drives. If only one disk drive is attached, there is no need to add a number after the device code D.

**Screen Editor:** (E:) Input and Output device. This device uses the keyboard and the display to simulate a screen-editing terminal. Writing to this device causes data to appear on the display starting at the current cursor position. Reading from this device activates the screen-editing process and allows the user to enter and edit data. Whenever the return key is pressed, the entire logical line within which the cursor resides is selected as the current record to be transferred to the user program.

**TV Monitor:** (S:) Input and Output device. This device allows the user to read characters from the display and write characters to it using the cursor as the screen addressing mechanism. Both text and graphics operations are supported.

**Interface, RS-232:** (R:) The RS-232 device enables the ATARI system to interface with RS-232-compatible devices such as printers, terminals, and plotters. It contains a port to which the 80-column printer (ATARI 825) can be attached.

## INPUT AND OUTPUT COMMANDS

The following commands are used to control input from the various peripherals and output to them. Because commands involving disk operations have been described in Chapter 7, they will not be covered again in this chapter.

### General Commands Controlling I/O Devices

OPEN(O.)

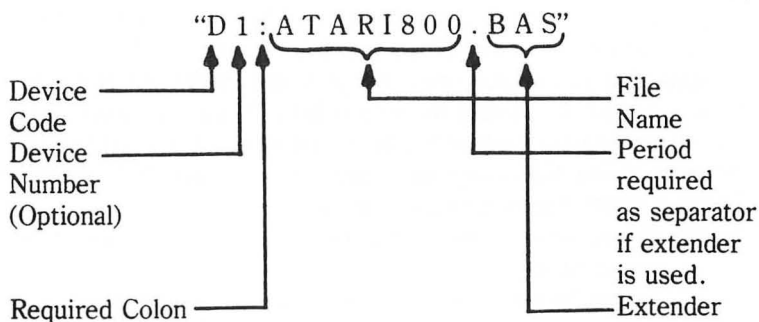
CLOSE(CL.)

Formats : OPEN #aexp, aexp1,aexp2,filespec  
CLOSE #aexpr

Examples: 100 OPEN #2,8,0,"D1:ATARI800.BAS"  
100 A\$="D1:ATARI800.BAS"  
110 OPEN #2,8,0,A\$  
150 CLOSE #2

Before a device can be accessed, it must be opened. This "opening" process links a specific IOCB to the appropriate device handler, initializes any CIO-related control variables, and passes any device-specific options to the device handler. The parameters for the OPEN command are defined as follows:

#	Mandatory character that must be entered by the user.
aexp	IOCB or file number. This number must be used with the same parameters in future operations. It may be any number from 1 through 7. It serves to select the device that will be utilized.
aexp1	Code number to determine input or output operation. Code    4 = input operation 8 = output operation 12 = input and output operation 6 = disk directory input operation 9 = end-of-file append (output) operation.  Append is also used for a special screen editor input mode. This mode allows a program to input the next line from E: without waiting for the user to press the return key.
aexp2	Device-dependent auxiliary code. An 83 in this parameter indicates sideways printing on a printer.
filespec	Specific file designation. Must be enclosed in quotation marks. The format for the file-spec parameter is shown below.



The close command simply closes files that have previously been opened with an open command. Note in the example that the

aexp number following the mandatory # character must be the same as the aexp reference number in the open statement.

Other commands which are most often associated with a single device, but which may be used to control other devices, are described on the following pages under the topics "Controlling Keyboard I/O" and "Controlling Output to the Screen"

## **Controlling Cassette Input and Output**

### **CLOAD(CLOA.)**

Format: CLOAD

Examples: CLOAD

100 CLOAD

This command can be used in either direct or deferred mode to load a program from cassette tape into RAM for execution. After you enter cload, one bell rings to indicate that you should press the play button and then the return key. However, do not press the play button until you have positioned the tape. Specific instructions for loading a program are contained in the ATARI 410 Program Recorder Manual. Steps for loading oversized programs are included in the paragraphs under "Chaining Programs" at the end of this section.

### **CSAVE(CS.)**

Format: CSAVE

Examples: CSAVE

100 CSAVE

100 CS.

This command is usually used in Direct mode to save a RAM-resident program onto cassette tape. Csave saves the tokenized version of the program. When you enter csave two bells ring to indicate that you should press the play and record buttons simultaneously and then press the return key. Do not, however, press these buttons until the tape has been positioned. It is faster to save a program using this command rather than a save "c" command because short interrecord gaps are used.

Notes: Tapes saved using the two commands, save and csave, are not compatible.

It may be necessary to enter an lprint before using csave.

Otherwise, csave may not work properly.

### **ENTER(E.)**



Format: ENTER filespec

Examples: ENTER "C"

This statement causes a program originally recorded using the list command to be loaded into RAM. The program is entered in unprocessed (untokenized) form, and is interpreted as the data is received. When the loading is complete, it may be run in the normal way. The enter command may also be used with the disk drive as outlined in Chapter 7. Note that both load and cload clear the old program from memory before loading the new one. Enter merges the old and new programs. This enter statement is usually used in direct mode.

## Controlling Keyboard I/O

### INPUT(I.)

Format: INPUT [#aexp {,}]{avar}{svar}[{avar}{svar}. . .]

Examples: 100 INPUT X  
100 INPUT N\$  
100 PRINT "ENTER THE VALUE OF X"  
100 INPUT X

This statement requests keyboard data from the user. The computer displays a ? prompt when the program encounters an input statement. The input statement is usually preceded by a print statement that tells the user what type of information is being requested. String variables are allowed only if they are not subscripted. Matrix variables are not allowed.

The #aexp (arithmetic expression) is optional and is used to specify the file or device number from which the data is to be input. (See your owner's manual for the specifics.) If no #aexp is specified, input is from the screen editor (E:). If several strings are to be input from the screen editor, you must type one string, press the return key, type the next string and press the return key again. Numerical values can be typed on the same line separated by commas.

## Controlling Output to the Printer

### LPRINT(LP.)

Format: LPRINT exp ; exp . . .

Example: LPRINT "PROGRAM TO CALCULATE X"  
100 LPRINT X; " ";Y;" ";Z

This statement causes the computer to print data on the line printer rather than on the screen. It can be used in either direct or deferred modes. It requires no device specifier and no open or close statement.

The following program listing illustrates a program that will add 5 numbers entered by the user.

```
10 PRINT "ENTER 5 NUMBERS TO BE SUMMED"
20 FOR N=1 TO 5
30 INPUT X
40 C=C+X
50 NEXT N
60 PRINT "THE SUM OF YOUR NUMBERS IS ";C
70 END
```

### Controlling Output to the Screen

PRINT(PR or ?)

Format: PRINT #aexp , exp ,exp . . .

Examples: PRINT X,Y,Z,A\$

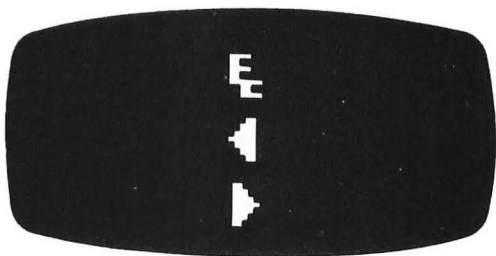
100 PRINT "THE VALUE OF X IS";X

100 PRINT "COMMAS", "CAUSE", "COLUMN",  
"SPACING"

100 PRINT #3,A\$

A print command can be used in either the direct or the deferred mode. In the direct mode, this command prints whatever information is contained between the quotation marks exactly as it appears. In the first example, print X,Y,Z,A\$, the screen will display the current values of X,Y,Z,A\$ as they appear in the RAM-resident program. In the last example, print #3,A\$, the 3 is the file specifier (may be any number between 1 and 7) that specifies the device where the value of A\$ will be printed.

A comma causes tabbing to the next tab location. Several commas in a row cause several tab jumps. A semicolon causes the next expression or value to be placed immediately after the preceding expression with no spacing. Therefore, in the second example, a space is placed before the ending quotation mark so the value of X will not be placed immediately after the word "IS". If no comma or semicolon is used at the end of a print statement, a return is output and the next print statement will start on the following line.



# Mathematical and Other Functions

The material in this chapter is designed to acquaint you with the arithmetic, trigonometric, and special purpose functions incorporated in the ATARI BASIC system. A function performs a computation and returns the result which can be either printed or used in additional computations. Included in the trigonometric functions are two statements, radians (RAD) and degrees (DEG), that are frequently used within trigonometric functions. Each function described in this chapter may be used in either the direct or the deferred mode. Multiple functions are perfectly legal.

The following functions and statements are described in this section:

ABS	RND	COS	FRE
CLOG	SGN	SIN	PEEK
EXP	SQR	DEG/RAD	POKE
INT	ATN	ADR	USR
LOG			

## ARITHMETIC FUNCTIONS

**ABS:** This function returns the absolute value of a number without regard to whether it is positive or negative. The returned value is always positive.

**CLOG:** Returns the logarithm to the base 10 equivalent of the

variable or expression in parentheses. CLOG(0) should give an error and CLOG(1) should be 0.

**EXP:** Returns the value of  $e$  raised to the power specified by the expression in parentheses. In some cases, EXP is accurate only to six significant digits.

**INT:** Returns the greatest integer less than or equal to the value of the expression. This is true whether the value of the expression is a positive or negative number. This INT function should not be confused with the function used on calculators that simply truncates (cuts off) all decimal places.

**LOG:** Returns the natural logarithm of the number or expression in parentheses. LOG(0) should give an error and LOG(1) should be 0.

**RND:** Returns a hardware-generated random number between 0 and 1, but never returns 1. The variable or expression in parentheses following RND is a dummy and has no effect on the numbers returned. However, the dummy variable must be used. Generally, the RND function is used in combination with other BASIC statements or functions to return a number for games, decision making, and the like. Here's a simple routine that returns a random number between 0 and 999.

```
10 X=RND(0)
20 RX=INT(1000*X)
30 PRINT RX
```

**SGN:** Returns a  $-1$  if the value of an expression is a negative number; a 0 if it is 0, or a 1 if it is a positive number.

**SQR:** Returns the square root of an expression whose value is positive.

## TRIGONOMETRIC FUNCTIONS

**ANT:** Returns the arctangent of the variable or expression in parentheses.

**COS:** Returns the trigonometric cosine of the expression in parentheses.

**SIN:** Returns the trigonometric sine of the expression in parentheses.

**DEG/RAD:** These two statements allow the programmer to specify degrees or radians for trigonometric function computations. The computer defaults to radians unless DEG is specified. Once the DEG statement has been executed, RAD must be used to return to radians.

## SPECIAL PURPOSE FUNCTIONS

**ADR:** Returns the decimal memory address of the string specified by the expression in parentheses. Knowing the address enables the programmer to pass the information to USR routines, etc.

**FRE:** This function returns the number of bytes of user RAM left. Its primary use is in the direct mode using a dummy variable (0) to inform the programmer how much memory space remains for completion of a program. Of course FRE can also be used with a BASIC program in the deferred mode.

**PEEK:** Returns the contents of a specified memory address location. The address specified must be an integer, or an arithmetic expression that evaluates to an integer, between 0 and 65535. It must represent the memory address in decimal notation. The number returned will also be a decimal integer with a range from 0 to 255. This function allows the user to examine either RAM or ROM locations. In the first example below, the peek is used to determine whether location 4000 (decimal) contains the number 255. In the second example below, the peek function is used to examine the left margin.

Format: PEEK(aexp)

Examples: 1000 IF PEEK (4000) = 255 THEN PRINT 255  
100 PRINT "LEFT MARGIN IS";PEEK (82)

**POKE:** Although this is not a function, it is included in this chapter because it is closely associated with the peek function. The poke command inserts data into the memory location or modifies data already stored there. Note that the data must be a decimal number between 0 and 255. Poke cannot be used to alter ROM locations. While you are gaining familiarity with this command, you should look at the memory location using a peek and write down the contents of the location. Then, if the poke doesn't work as anticipated, the original contents can be poked back into the location.

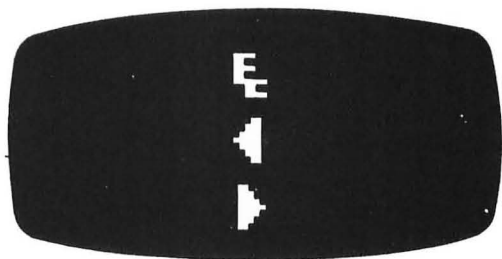
**USR:** This function returns the results of a machine-language subroutine. The first expression (aexpl) must be an integer, or an arithmetic expression whose value is an integer, that represents the decimal memory address of the machine language routine to be performed. Subsequent input arguments, aexp2, aexp3, etc., are optional. These should be arithmetic expressions within a decimal range of 0 through 65535. If a noninteger value is used it will be rounded off to the nearest integer.

N	(Number of arguments on the stack-may be 0)
X <sub>1</sub>	(High byte of argument X)
X <sub>2</sub>	(Low byte of argument X)
Y <sub>1</sub>	(High byte of argument Y)
Y <sub>2</sub>	(Low byte of argument Y)
Z <sub>1</sub>	(High byte of argument Z)
Z <sub>2</sub>	(Low byte of argument Z)
.	
.	
R <sub>1</sub>	(Low byte of return address)
R <sub>2</sub>	(High byte of return address)

Fig. 9-1. Hardware stack definition.

These values returned will be converted from BASIC's Binary Coded Decimal floating point number format to a two-byte binary number; then pushed onto the hardware stack. The hardware stack is composed of a group of RAM locations under the direct control of the 6502 microprocessor chip. Figure 9-1 illustrates the structure of the hardware stack.

## Chapter 10



### Programming Techniques to Save Memory

This chapter includes hints on increasing programming efficiency and conserving memory. These hints give ways of conserving memory on the ATARI 800. Some of these methods make programs less readable and harder to modify, but there are cases when this is necessary due to memory limitations.

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 800, which automatically removes extra spaces. Statements are always displayed in the same manner no matter how many spaces were used on program entry. Spaces should be used (just as in typing on a conventional typewriter) between successive words and between words and variable names. Here is an example:

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

Note the space 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. Therefore use conventional spacing.

2. Each new line number represents the beginning of what is called a new "logical line". Each logical line takes 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 takes only 3 bytes. If you need to save memory, avoid programs like the following:

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

and consolidate lines like this:

```
10 X=X+1:Y=Y+1:Z=X+Y:PRINT Z: GOTO 10
```

The above consolidation saves 12 bytes.

3. Variables and constants should be “managed” for savings, too. Each time a constant (4,5,16,3.14, etc.) is used, it takes 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 is used more than once or twice in a program, it should be defined as a variable, and the variable name should be used throughout the program. For example:

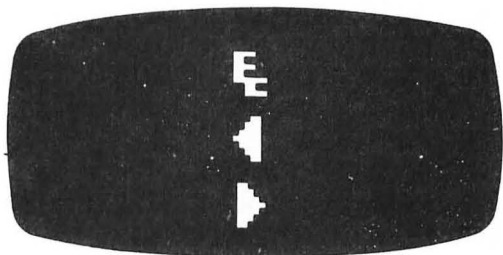
```
10 PI=3.14159
20 PRINT “AREA OF A CIRCLE IS THE RADIUS
   SQUARED TIMES ”;PI
```

4. Literal strings found in print statements require 2 bytes overhead and 1 byte for each character (including all spaces) in string.

5. String variables take 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 when it is defined. Obviously, the use of string variables is very costly in terms of RAM.

6. 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. Thus, a 25 row by 4 column matrix would require 15 + (approximately) 3 + (approximately) 10+6 times 100, or about 630 bytes.





## Sample Programs: Conversions

In this day and age conversions are quite common . . . Celsius to Fahrenheit, centimeters to inches, dollars to pesos . . . just to name a few. The ATARI can be used to help you make these conversions quickly and easily. The following programs are examples of several such conversions. Other uses for the programs should readily come to mind as you read through these.

### CURRENCY EXCHANGE

More and more Americans are traveling to foreign countries each year, and as you know, monetary values are different. Let's assume that you plan a trip to Britain and France and you wish to convert monetary values among three systems: British pound, French franc, and U.S. dollar. In order to facilitate these conversions, you decide to write a program to compute them for you. Each currency system is to be denoted by a code number. You must initialize the program each time you use it by entering equivalent monetary values for each currency. Then you can enter the currency code and the amount to be converted. The currency system and the equivalent amount will be printed.

On a certain morning, 1 British pound is equivalent to 8.3981 francs and 1.8248 U.S. dollars. At these rates, find how much an airplane ticket worth 284 British pounds will cost in dollars and in francs. Do the same for an English double-barreled shotgun valued at 1205 U.S. dollars.

```

10 REM* CURRENCY EXCHANGE
20 PRINT "ENTER CODE, AMT FOR 3
   SYSTEMS"
30 PRINT "1=BR, 2=FR, 3=US"
40 INPUT C1,A1,C2,A2,C3,A3
50 PRINT
60 PRINT "CODE, AMT TO CONVERT (
   0,0=STOP)"
70 INPUT C,A
80 IF C=0 THEN STOP
90 REM* DETERMINE REFERENCE
100 ON C GOTO 110,130,150
110 R=A1
120 GOTO 160
130 R=A2
140 GOTO 160
150 R=A3
160 V1=A*A1/R
170 V2=A*A2/R
180 V3=A*A3/R
190 PRINT "EQUIVALENT AMOUNTS:"
200 PRINT "BR. POUND ";V1
210 PRINT "R.F. FRANC ";V2
220 PRINT "U.S. DOLLAR ";V3
230 PRINT
240 GOTO 50
250 END

```

### Display

When the following is run, you will see the following on your screen:

```

ENTER CODE, AMT FOR 3 SYSTEMS
1=BR, 2=FR, 3=US
?
1;1,2,8.3981,3,1.8248
CODE, AMT TO CONVERT (0,0=STOP)
?
1,284
EQUIVALENT AMOUNTS
BR. POUND 284
R.F. FRANC 2385

```

U.S. DOLLAR 518  
 CODE, AMT TO CONVERT (0,0=STOP)  
 ?  
 3,1205  
 EQUIVALENT AMOUNTS  
 BR. POUND 669  
 R.F. FRANC 5622  
 U.S. DOLLAR 1205  
 CODE, AMT TO CONVERT (0,0=STOP)  
 ?  
 0,0

## METRIC CONVERSIONS

The metric system will eventually become universal—or so it seems at this time—and the ATARI can be programmed to make these conversions for you. The program which follows allows conversions of length, area, mass (weight), and liquid volume. The conversion can be either to or from the metric units.

```

10  REM *METRIC CONVERSION PROGRAM
20  M=8
30  M1=9
40  M2=6
50  M3=8
60  DIM L1$(8)
70  DIM L2$(8)
80  DIM L3(8)
90  DIM A1$(9)
100 DIM A2$(9)
110 DIM A3(9)
120 DIM W1$(6)
130 DIM W2$(6)
140 DIM W3(6)
150 DIM V1$(8)
160 DIM V2$(8)
170 DIM V3(8)
180 FOR I = 1 TO M
190   READ LI$(I),L2$(I),L3(I)
200 NEXT I
210 FOR I = 1 TO M1
220   READ A1$(I),A2$(I),A3(I)
230 NEXT I
  
```

```

240 FOR I = 1 TO M2
250   READ W1$(I),W2$(I),W3(I)
260 NEXT I
270 FOR I = 1 TO M3
280   READ V1$(I),V2$(I),V3(I)
290 NEXT I
300 REM *****
310 REM **** PROCESSING AREA ****
320 PRINT"DO YOU WISH TO CONVERT LENGTH (L),
    AREA (A), "
330 PRINT"          MASS (M), OR LIQUID VOLUME (V)?"
340 INPUT A1$
350 PRINT
360 PRINT:          CONVERSIONS AVAILABLE"
370 PRINT
380 PRINTTAB(9);"NER";TAB(14);"
    FROM";TAB(39);" TO"
390 PRINTTAB(9);"---";TAB(14);"-----";TAB(39);"-----"
400 IF A1$="L" THEN 460
410 IF A1$="A" THEN 590
420 IF A1$="M" THEN 720
430 IF A1$="V" THEN 850
440 PRINT"ENTRY MUST BE L, A, M, OR V"
450 GOTO 300
460 REM **** PRINT FOR LENGTH ****
470 FOR I = 1 TO M
480   PRINTAB(10);I;TAB(15);L1$(I);TAB(40);L2$(I)
490 NEXT I
500 PRINT
510 PRINT"ENTER THE NUMBER OF THE CONVERSION
    TO BE USED (0 WHEN DONE)";
520 INPUT N
530 IF N=0 THEN 980
540 PRINT"ENTER THE NUMBER OF ";L1$(N)
550 INPUT L0
560 L=L0*L3(N)
570 PRINTL0;L1$(N);"="L;L2$(N)
580 GOTO 500
590 REM **** PRINT OF AREA ****
600 FOR I = 1 TO M1
610   PRINTTAB(9);I;TAB(14);A1$(I);TAB(39);A2$(I)
620 NEXT I

```

```

630 PRINT
640 PRINT"ENTER THE NUMBER OF THE CONVERSION
    TO BE USED (0 WHEN DONE)"
650 INPUT N
660 IF N=0 THEN 990
670 PRINT"ENTER THE NUMBER OF ";A1$(N);
680 INPUT A0
690 A=A0*A3(N)
700 PRINTA0;A1$(N);"=";A;A2$(N)
710 GOTO 630
720 REM ***** PRINT OF MASS *****
730 FOR I = 1 TO M2
740     PRINTTAB(10);I;TAB(15);W1$(I);TAB(40);W2$(I)
750 NEXT I
760 PRINT
770 PRINT"ENTER THE NUMBER OF THE CONVERSION
    TO BE USED (0 WHEN DONE)";
780 INPUT N
790 IF N=0 THEN 990
880 PRINT"ENTER THE NUMBER OF ";W1$(N);
810 INPUT W0
820 W=W0*W3(N)
830 PRINTW0/W1$(N); "=";W;W2$(N)
840 GOTO 760
850 REM **** PRINT OF LIQUID VOLUME ****
860 FOR I = 1 TO M3
870     PRINTTAB(10);I;TAB(15);V1$(I);TAB(40);V2$(I)
880 NEXT I
890 PRINT
900 PRINT"ENTER THE NUMBER OF THE CONVERSION
    TO BE USED (0 WHEN DONE)";
910 INPUT N
920 IF N=0 THEN 990
930 PRINT"ENTER THE NUMBER OF ";V1$(N);
940 INPUT V0
950 V=V0*V3(N)
960 PRINTV0;V1$(N);"=";V;V2$(N)
970 GOTO 890
980 REM *****
990 REM ***** PROGRAM TERMINATION POINT *****
1000 PRINT
1010 PRINT "END"

```

```

1020 END
1030 REM *****
1040 REM ***** DATA FOR INITIALIZATION *****
1050 REM ***** LENGTH *****
1060 DATA INCHES, MILLIMETERS, 25.4, FEET, METERS, .3048
1070 DATA YARDS, METERS, .0144, MILES, KILOMETERS, 1.6093
1080 DATA MILLIMETERS, INCHES, .0394, METERS, FEET, 3.2808
1090 DATA METERS, YARDS, 1.0936, KILOMETERS, MILES, .6214
1100 REM ***** AREA
1110 DATA SQ INCHES, SQ CENTIMETERS, 6.4516, SQ FEET, SQ METERS, .0929
1120 DATA SQ YARDS, SQ METERS, .8361, SQ MILES, SQ KILOMETERS, 2.59
1130 DATA ACRES, SQ HECTOMETERS (HECTARES), .4047
1140 DATA SQ CENTIMETERS, SQ INCHES, .115, SQ METERS, SQ YARDS, 1.196
1150 DATA SQ KILOMETERS, SQ MILES, .3861
1160 DATA SQ HECTOMETERS (HECTARES), ACRES, 2.471
1170 REM ***** MASS
1180 DATA OUNCES, GRAMS, 28.3495, POUNDS, KILOGRAMS, .4536
1190 DATA SHORT TONS, MEGAGRAMS, .9, GRAMS, OUNCES, .0353
2000 DATA KILOGRAMS, POUNDS, 2.2046, MEGAGRAMS, SHORT TONS, 1.1
1200 REM ***** LIQUID VOLUME
1220 DATA OUNCE, MILLILITERS, 30, PINTS, LITERS, .4732
1230 DATA QUARTS, LITERS, .9464, GALLONS, LITERS, 3.7856
1240 DATA MILLILITERS, OUNCES, .03, LITERS, PINTS, 2.1134
1250 DATA LITERS, QUARTS, 1.0567, LITERS, GALLONS, .2642
1260 REM *****

```

**Display:** When the program is run, you may see the following on your screen:

DO YOU WISH TO CONVERT LENGTH (L), AREA (A),  
MASS (M), OR LIQUID  
VOLUME (V)?

?L

CONVERSIONS AVAILABLE		
NBR	FROM	TO
1	INCHES	MILLIMETERS
2	FEET	METERS
3	YARDS	METERS
4	MILES	KILOMETERS
5	MILLIMETERS	INCHES
6	METERS	FEET
7	METERS	YARDS
8	KILOMETERS	MILES

ENTER THE NUMBER OF THE CONVERSION TO BE USED (0  
WHEN DONE)? 3

ENTER THE NUMBER OF YARDS? 10

10 YARDS= 9.144 METERS

ENTER THE NUMBER OF THE CONVERSION TO BE USED (0  
WHEN DONE)? 0

END

If you run the program again and do a different conversion, you  
may see the following:

DO YOU WISH TO CONVERT LENGTH (L), AREA(A),  
MASS (M), OR LIQUID VOL-  
UME (V)?

?A

CONVERSIONS AVAILABLE		
NBR	FROM	TO
1	SQ INCHES	SQ CENTIMETERS
2	SQ FEET	SQ METERS
3	SQ YARDS	SQ METERS
4	SQ MILES	SQ KILOMETERS
5	ACRES	SQ HECTOMETERS (HECTARES)
6	SQ CENTIMETERS	SQ INCHES
7	SQ METERS	SQ YARDS
8	SQ KILOMETERS	SQ MILES
9	SQ HECTOMETERS(HECTARES)	ACRES

ENTER THE NUMBER OF THE CONVERSION TO BE USED (0  
WHEN DONE)? 4

ENTER THE NUMBER OF SQUMILES? 10

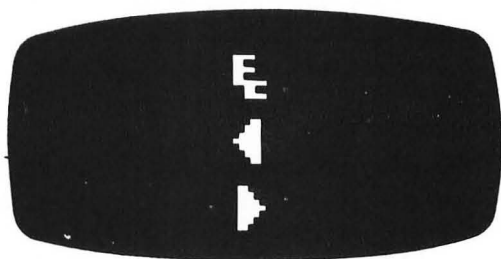
10 SQ MILES=25.9 SQ KILOMETERS

ENTER THE NUMBER OF THE CONVERSION TO BE USED (0  
WHEN DONE)? 0

END



## Chapter 12



### More Programs

The following programs in BASIC can be run as they are for experience and then modified to suit your own personal needs. For example, the one on gun collecting could be modified to be used to organize a coin collection . . . or perhaps your bottle or old canning jar collection.

To give you some experience with you ATARI 800, some of the programs will require a slight modification to make them work. Can you make them work?

#### **FINDING GREATEST COMMON DIVISOR OF TWO NUMBERS**

##### **Description**

This program finds the greatest common divisor of two numbers.

##### **Functions of the Program**

The program reads in the number of greatest common divisors to be found and then the pairs of numbers for which the greatest common divisors are desired. You, however will have to supply the lines that will determine the greatest common divisor.

##### **Instruction for Use**

Use data statements to supply the number of pairs of numbers for which a G.C.D. is desired, followed by those pairs of numbers.

## Data Format

The format to put the data in is:

1. Number of greatest common divisors needed.
2. First number of first pair.
3. Second number of first pair.
4. First number of second pair, etc.

## Output Description

Appropriate headings with columns of supplied data and the calculated G.C.D. (if you add the right lines).

```
10 REM This program finds the greatest common divisor of 2
   numbers
15 REM N is the number of pairs of numbers
20 REM A and B are the two numbers for which a G.C.D. is
   desired
25 REM Q is the quotient and R is the remainder of A/B
30 READ N
35 PRINT "NUMBER OF G.C.D. to be found is ",N
37 PRINT
40 IF N=0 THEN 99
45 PRINT "A", "B", "GCD"
50 READ A,B
55 PRINT A,B;
60 LET Q = INT (A/B)
65 LET R=A-Q*B
70 IF R=0 THEN 90
75 LET A=B
80 PRINT B
85 GOTO 60
90 PRINT B
92 LET N=N-1
94 IF N=0 THEN 99
95 GOTO 50
97 DATA 3,256,243,5,15,130,169
99 END
```

RUN

Number of G.C.D.s to be found is 3

A	B	G.C.D.
256	243	1

5	15	5
130	169	13

## **PROGRAM FOR GUN COLLECTORS**

### **Description**

This program enables gun collectors to organize and control their collections. All the items in your collection can be listed, or you can select items by the manufacturer.

### **Functions of the Program**

The program accepts the gun information from the data statements and prints the items specified. Items to be printed can be selected according to manufacturer or the entire list can be printed. Watch out! Some of the program lines will have to be fixed. A few hints: Are the same variable names used for different variables? Are variable names changed or confused? Are string variables used where necessary? Are there references to nonexistent line numbers?

### **Instructions for Use**

Use data statements to enter the total number of guns to be listed and to enter the pertinent information concerning each gun.

### **Data Format**

The format to put the data in is: manufacturer, model number, serial number, type of action, caliber or gauge, condition, value.

### **Output Description**

See the example below. Output is either a formatted list of all items or a list of those items that satisfy the selection criteria.

### **Important Variables**

M	..Maximum number of data reads (total number of guns to be entered)
M2	..Manufacturer of gun
N	..Model number
S	..Serial number
C	..Caliber or gauge
A	..Type of action
C2	..Condition
V	..Current Value

```

10 REM GUN COLLECTION PROGRAM
20 REM *** READ DATA ***
30 PRINT "ENTER TOTAL NUMBER OF GUNS TO BE
   RECORDED"
40 INPUT M
50 PRINT "DO YOU WISH TO HAVE ALL ENTRIES
   LISTED? (Y OR N)"
60 INPUT L$
70 PRINT
80 IF L$ = "Y" THEN 110
90 PRINT "ENTER MANUFACTURER YOU WISH LISTED"
100 INPUT L0
110 PRINT
120 PRINT
130 PRINT "MANUFACTURER", TAB(15); "Model"
   TAB(10); "Serial";
140 PRINT TAB(10); "Caliber"; TAB(10); "Action"; TAB(10);
150 PRINT "Condition"; TAB(10); "Value"
155 PRINT
160 FOR I=1 TO M
170 READ I1$
180 IF I1$ = "END" THEN 320
190 READ M2,N,S,C,A,C2,V
200 IF Z$ < > "Y" THEN 240
210 PRINT I1$; TAB(10); A; TAB(15); C2; TAB(10); V
230 GOTO
240 IF I1$ < > A0 THEN 310
250 C=C+1
260 IF C=1 THEN 290
270 I1$=" "
280 PRINT I1$; TAB (15);M2; TAB(10);S;TAB(10);
300 PRINT C; TAB(10); A; TAB(15);C2; TAB(10);V
310 NEXT I
320 PRINT
330 PRINT
340 STOP
350 REM *** DATA ***
360 DATA Winchester, 70, 44271, .270W,Bolt,Good, 450
370 DATA Remington, 1100, 00714, 12, Auto, Ex, 312
380 DATA H and R, 158C, 77431, 20, single, good, 95
390 DATA Remington, 742, 00853, .30-06, auto, ex, 375
400 DATA END

```

RUN

ENTER TOTAL NUMBER OF GUNS TO BE RECORDED ?4

DO YOU WISH TO HAVE ALL ENTRIES LISTED? (Y OR N) ? Y

Manufacturer	Model	Serial	Caliber	Action	Condition	Value
Winchester	70	44371	.270W	Bolt	Good	450
Remington	1100	00714	12	Auto	Ex	312
H and R	158C	77431	20	Single	Good	95
Remington	742	00853	30-06	Auto	Ex	375

RUN

ENTER MAXIMUM NUMBER OF GUNS TO BE RECORDED

?4

DO YOU WISH TO HAVE ALL ENTRIES LISTED? (Y OR N) N

?Remington

Manufacturer	Model	Serial	Caliber	Action	Condition	Value
Remington	1100	00714	12	Auto	Ex	312
	742	00853	30-06	Auto	Ex	375

## CHECKBOOK BALANCER

This is one of the "traditional" programs that every beginning computerist writes. It allows the entry of outstanding checks and uncredited deposits as well as cleared checks and credited deposits. Remember that the question mark is simply a symbol for print.

```
10 DIM A$(30),MSG$(40),MSG1$(30),MSG2$(30)
   ,MSG3$(30),MSG4$(30),MSG5$(30),MSG6$(30)
20 OUTSTAND=0
30 GRAPHICS 0:?:? " CHECKBOOK BALANCER":?
40 ? "You may make corrections at any time by entering a
   negative dollar value."
50 MSG1$="OLD CHECK -- STILL OUTSTANDING"
60 MSG2$="OLD DEPOSIT -- NOT CREDITED"
70 MSG3$="OLD CHECK -- JUST CLEARED"
80 MSG4$="OLD DEPOSIT -- JUST CREDITED"
90 MSG5$="NEW CHECK (OR SERVICE CHARGE)"
100 MSG6$="NEW DEPOSIT (OR INTEREST)"
150 TRAP 150:?: "Enter beginning balance from your
   checkbook";INPUT YOURBAL
160 TRAP 160:?: "Enter beginning balance from your
   bankstatement";INPUT BANKBAL
165 TRAP 40000
170 GOTO 190
180 CLOSE #1:?: "PRINTER IS NOT OPERATIONAL."
```

```

185 ? "PLEASE CHECK CONNECTORS."
190 PERM=0
200 ? "Would you like a permanent record on the printer";:IN-
    PUT A$
210 IF LEN(A$)=0 THEN 200
220 IF A$(1,1)="N" THEN 400
230 IF A$(1,1) "Y" THEN 200
240 TRAP 180
250 LPRINT :REM TEST PRINTER
260 PERM=1
280 LPRINT "YOUR BEGINNING BALANCE IS $";YOUR-
    BAL
290 LPRINT "BANK STATEMENT BEGINNING BALANCE
    IS $";BANKBAL:LPRINT
400 TRAP 400:? :? "Choose one of the following:"
410 ? "(1) ";MSG1$
415 ? "(2) ";MSG2$
420 ? "(3) ";MSG3$
425 ? "(4) ";MSG4$
430 ? "(5) ";MSG5$
435 ? "(6) ";MSG6$
440 ? "(7) ";NO MORE ENTRIES"
490 ?
500 INPUT N:IF N<1 OR N>7 THEN 400
505 TRAP 40000
510 ON N GOSUB 1000,2000,3000,4000,5000,6000,7000
520 MSG$="NEW CHECKBOOK BALANCE IS": AMOUNT
    =YOURBAL:GOSUB 8000
530 MSG$="NEW BANK STATEMENT BALANCE
    IS":AMOUNT=BANKBAL:GOSUB 8000
540 MSG$ = "OUTSTANDING CHECKS - DEPOSITS="
    :AMOUNT=OUTSTAND:GOSUB 8000
545 IF PERM THEN LPRINT
550 GOTO 400
1000 REM OLD CHECK -- STILL OUTSTANDING
1010 MSG$=MSG1$:GOSUB 8100
1020 OUTSTAND=OUTSTAND+AMOUNT
1030 RETURN
2000 REM OLD DEPOSIT -- STILL NOT CREDITED
2010 MSG$=MSG2$:GOSUB 8100
2020 OUTSTAND=OUTSTAND-AMOUNT

```

```

2030 RETURN
3000 REM OLD CHECK - - JUST CLEARED
3010 MSG$=MSG3$:GOSUB 8100
3020 BANKBAL=BANKBAL-AMOUNT
3030 RETURN
4000 REM OLD DEPOSIT -- JUST CREDITED
4010 MSG$=MSG4$:GOSUB 8100
4020 BANKBAL=BANKBAL+AMOUNT
4030 RETURN
5000 REM NEW CHECK (OR SERVICE CHARGE) -- JUST
      CLEARED
5010 MSG$=MSG5$:GOSUB 8100
5020 YOURBAL=YOURBAL-AMOUNT
5030 ? "IS NEW CHECK STILL OUTSTANDING";:INPUT A$
5040 IF LEN(A$)=0 THEN 5030
5050 IF A$(1,1) "N" THEN 5060
5055 BANKBAL=BANKBAL-AMOUNT
5057 IF PERM THEN LPRINT "CHECK HAS CLEARED."
5058 RETURN
5060 IF A$(1,1) < > "Y" THEN 5030
5070 OUTSTAND=OUTSTAND+AMOUNT
5075 IF PERM THEN LPRINT "CHECK IS STILL OUT-
      STANDING."
5080 RETURN
6000 REM NEW DEPOSIT (OR INTEREST) -- JUST CRE-
      DITED
6010 MSG$=MSG6$:GOSUB 8100
6020 YOURBAL=YOURBAL+AMOUNT
6030 ? "HAS YOUR NEW DEPOSIT BEEN CREDITED";:IN-
      PUT A$
6040 IF LEN(A$)=0 THEN 6030
6050 IF A$(1,1) < > "Y" THEN 6060
6052 BANKBAL=BANKBAL+AMOUNT
6053 IF PERM THEN LPRINT "DEPOSIT HAS BEEN CRE-
      DITED."
6055 RETURN
6060 IF A$(1,1) < > "N" THEN 6030
6070 OUTSTAND=OUTSTAND-AMOUNT
6075 IF PERM THEN LPRINT "DEPOSIT HAS NOT BEEN
      CREDITED."
6080 RETURN

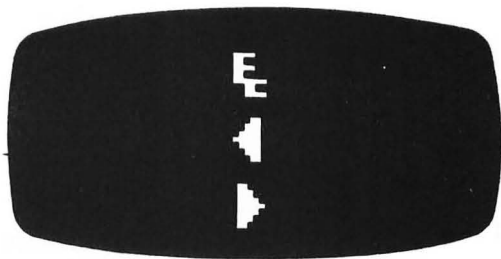
```

```

7000 REM DONE
7010 ? "BANK'S BALANCE MINUS (OUTSTAND-
      ING CHECKS-DEPOSITS) SHOULD NOW EQUAL
      YOUR CHECKBOOK BALANCE."
7020 DIF=YOURBAL-(BANKBAL-OUTSTAND)
7030 IF DIF<>0 THEN 7040
7035 ? "IS $";BANKBAL;" THE ENDING BALANCE ON YOUR
      BANK STATEMENT";:INPUT A$
7036 IF LEN(A$)=0 THEN 7035
7037 IF A$(1,1)="Y" THEN ? CONGRATULATIONS: YOUR
      CHECKBOOK BALANCES!":END
7038 GOTO 7060
7040 IF DIF>0 THEN ? "YOUR CHECKBOOK TOTAL IS
      $";DIF;" OVER YOUR BANK'S TOTAL. ":GOTO 7060
7050 ? "WOULD YOU LIKE TO MAKE CORRECTIONS?"
7070 ? "REMEMBER, YOU CAN ENTER A NEGATIVE DOL-
      LAR VALUE TO MAKE A CORRECTION."
7080 ? "ENTER Y OR N";:INPUT A$
7090 IF LEN(A$)=0 THEN END
7100 IF A$(1,1)="Y" THEN RETURN
7110 END
7999 REM MSG PRINTING ROUTINE
8000 ? MSG$;" $";AMOUNT
8010 IF PERM=1 THEN LPRINT MSG$;" $";AMOUNT
8020 RETURN
8100 REM MSG PRINT & INPUT ROUTINE
8110 TRAP 8110:? "ENTER AMOUNT FOR ";MSG$;:INPUT
      AMOUNT
8120 TRAP 40000
8130 IF PERM=1 THEN LPRINT MSG$;" $";AMOUNT

```





### Flowcharting Techniques

If you are a beginning programmer, you may be interested in learning some of the techniques that allow you to more easily advance a programming task from a conceptual stage to a finalized stage.

Flowcharts, which pictorialize the solution to a programming task, are valuable programming tools. They are often drawn long before the actual program statements are written.

While flowcharting your program, you might change or simplify your approach, or see a flaw in your logic. After several attempts, you should have a workable flowchart and, once you do, your programming task is greatly reduced.

Any flowchart that you draw is useful; but a few basic conventions are described in this chapter. Terminal (that is, starting or ending) activities are represented by ovals. Arrows, indicate the direction of program flow between operations. Most calculator operations are represented by rectangles. A diamond represents a decision point.

To illustrate, in the following example we have labeled each flowcharting operation with the corresponding program line number. See Fig. 13-1. As you can see, once the flowchart is finalized, the program can be written relatively easily.

- 10 REM—THIS PROGRAM AVERAGES UP TO 20 POSITIVE NUMBERS
- 20 REM—IF YOU HAVE FEWER THAN 20 NUMBERS TO

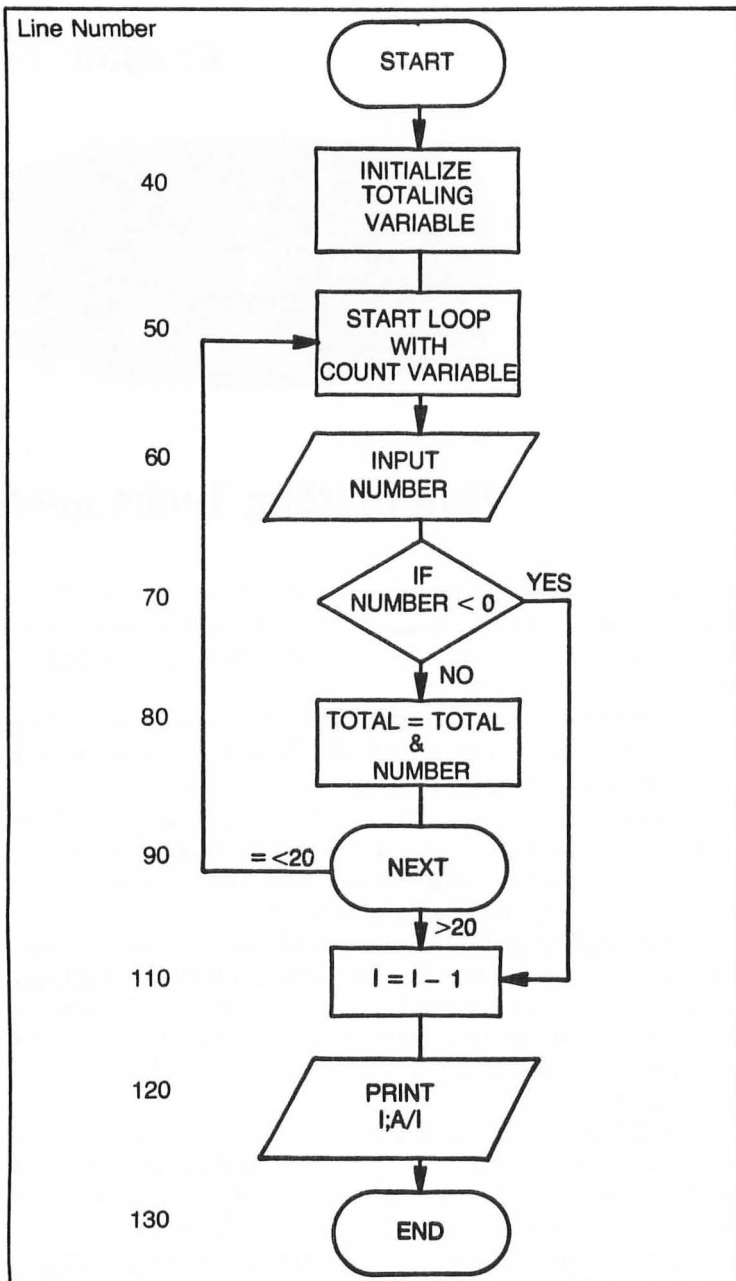


Fig. 13-1.

```

ENTER.
30 REM—ENTER A NEGATIVE NUMBER TO END THE
  INPUTTING.
40 A=0
50 FOR I=1 TO 20
60 INPUT B
70 IF B <0 THEN 110
80 A=A+B
90 NEXT I
110 I=I-1
120 PRINT "THE AVERAGE OF THE "I" NUMBERS IS "A/I
130 END

```

As mentioned previously, flowcharts use boxes and lines, following a few simple rules, to represent a program design. Only a few shapes and a few ways of drawing the connecting lines are used. These restrictions are intended to make the resulting flowcharts easy to understand by displaying the relationship of parts very clearly. After a correct flowchart has been drawn, translation to a program (writing statements corresponding to the boxes) is relatively simple.

## FLOWCHARTING SYMBOLS

Five different shapes of boxes are required for most flowcharts.

### Rectangle

A rectangle indicates any processing operation except input/output or a decision.

### Diamond

A diamond indicates a decision. The lines leaving the box are labeled with the results that cause each path to be followed.

### Parallelogram

A parallelogram indicates an input or output operation.

### Oval

An oval indicates the beginning or ending point of a program or program segment.

## **Circle**

A small circle indicates a collection point, where lines from other boxes join.

## **CONTROL STRUCTURES**

The fundamental function of a flowchart, as of any other program design tool, is to show in exact and unambiguous detail the sequence in which actions are to be carried out and the conditions under which alternatives are to be taken. In depicting these matters in flowcharts, three control structures are normally used.

### **Sequence**

When an arrow leads from one box to the next, that means simply that the two are to be executed in sequence in the order shown.

### **Selection**

The choice indicated by the decision box always has just two outcomes, depending on whether the condition is true or false.

### **Iteration**

Processing actions are executed repeatedly until a stated condition terminates the repetition. The number of repeats may in some cases be zero.

The basic structures stated above may be combined. For example, a path coming out of a decision box may lead to another decision or perhaps an iteration. Furthermore, it will commonly happen that the actions specified in a loop will include decisions. This is perfectly acceptable so long as only the three basic structures are used.

## **THE IF STATEMENT**

Often there are times when you want to make a decision. In the averaging program discussed previously, we wanted the program to branch to either the end of the program, or to the inquiry for more information. The branch was dependent on the outcome of a specified condition, using the if . . . then statement.

IF numeric expression THEN statement

The if . . . then statement makes a decision based upon the outcome of the numeric expression. If the expression is true, the then part of the statement is executed. If the outcome is false,

execution continues with the statement following the if . . . then statement.

For example, suppose an accountant wishes to write a program that will calculate and print the amount of tax to be paid by a number of persons. For those with incomes of \$10,000 per year or less, the amount of tax is 17.5%. For those with incomes of over \$10,000, the tax is 20%. A flowchart for the program might look like the one in Fig. 13-2. The diamond in the flowchart would be represented by an if . . . then statement in a program. The program may look like the following:

```
10 PRINT "INCOME"
20 INPUT I
30 IF I > 10000 THEN 60
40 PRINT "TAX ="; I*.175
50 GOTO 70
60 PRINT "TAX="; I*.2
70 END
```

As you can see, we used a relational operation in the if . . . then statement. The if . . . then statement is most often used with relational operators, although the decision can be based on the value of any numeric expression.

If the condition is true—the income is greater than \$10,000—program control is transferred to statement 60. If the condition is false—the income is less than or equal to \$10,000—the rest of the if statement is ignored and the program continues at statement 40. To test this program, use the values of \$20,000 and \$9,000.

## THE ELSE OPTION

The ATARI does not have the else option, however its action is useful to study while discussing flowcharts. In the previous example, if the numeric expression was evaluated as false, program execution continued with the next sequential statement following the if . . . then statement. But if you specify the else option with the if . . . then statement, the program will instead perform the instructions that follow the else. This gives you tremendous power using conditional branching.

IF numeric expression THEN state number or executable instr.  
ELSE statement number or executable instr.

If the numeric expression is false and else is specified execution is transferred to the statement number following else or the

Line Number

10

START

20

PRINT  
PROMPT

30

INPUT  
INCOME

40

IS  
INCOME >  
10,000  
?

YES

50

OUTPUT  
INCOME \*  
.175

GOTO  
END

60

OUTPUT  
INCOME \*  
.2

70

END

Fig. 13-2.

indicated else statement is executed. Look at the following example.

A quadratic equation is of the form  $0=ax^2+bx+c$ . If  $a=0$ , its two roots may be found by the formulas

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

To write a program to compute the roots of a quadratic equation given the values of the coefficients  $a$ ,  $b$ , and  $c$ , proceed as follows: If  $a$  is zero, display an error message and reenter new values. If  $b^2 - 4ac$  is less than zero, then the square root of that value would give a warning message or an error. So make sure that  $b^2 - 4ac$  is greater than or equal to zero before you compute the roots. The flowchart is shown in Fig. 13-3. In this solution we use two forms of the if . . . then . . . else statement. Study it carefully and then load the program and run it.

```
10 REM*ROOTS*
20 PRINT "IF A QUADRATIC"
30 PRINT "EQUATION IS OF THE"
40 PRINT "FORM 0=A*X 2+B*X+C"
50 PRINT "ENTER A, B, C"
60 INPUT A, B, C
70 D=B*B-4*A*C
80 IF A=0 THEN PRINT "A=0; NOT QUADRATIC.
   REENTER VALUES" ELSE 100
90 GOTO 60
100 IF D >=0 THEN 120 ELSE PRINT "COMPLEX ROOTS:
   CANNOT COMPUTE. REENTER VALUES."
110 GOTO 60
120 R1=(-B+SQR(D))/(2*A)
130 R2=(-B-SQR(D))/(2*A)
140 PRINT "COEFFICIENTS=";A;B;C
150 PRINT "ROOTS=";R1;R2
160 END
```

To summarize, if  $A = 0$ , the program displays message, then continues to next statement. If  $A = 0$ , the statement else 100 is read and the program branches to statement 100. If  $D \geq 0$ , the program branches to line 120. If  $D < 0$ , the else message is displayed and the program then continues to statement 110.

Run the program to find the roots of the equation  $x^2+x-6=0$ . Then run the program again to test the decision with  $x+1=0$  and

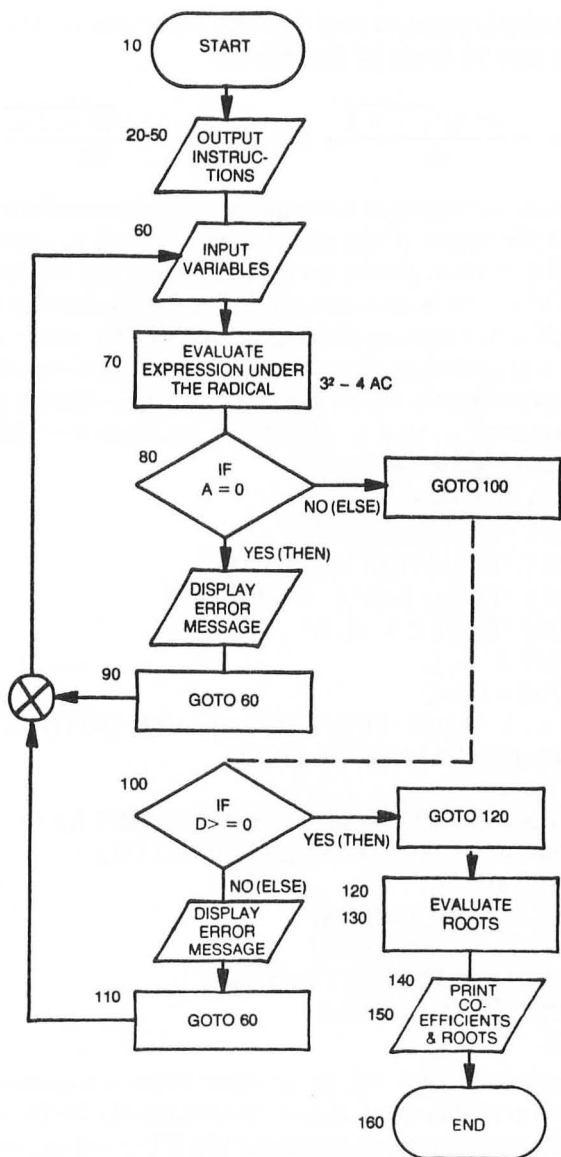


Fig. 13-3.



$$x^2+2x-1=0$$

```

RUN
IF A QUADRATIC EQ
EQUATION IS OF THE
FORM  $0=A*X^2+B*X+C$ 
ENTER A, B, C
?
1, 1, -6 ( $x^2 + x - 6$ )
COEFFICIENTS = 1 1 - 6
ROOTS=2 -3
RUN
IF A QUADRATIC
EQUATION IS OF THE
FORM  $0=A*X^2+B*X+C$ 
ENTER A, B, C
?
1, 2, 2 ( $X^2 + 2 X + 2$ )
COMPLEX ROOTS: CANNOT COMPUTE
REENTER VALUES
?
3, 2, -1 ( $3X^2 + 2X - 1$ )
COEFFICIENTS=3 2 -
ROOTS= .333333333333 - 1

```

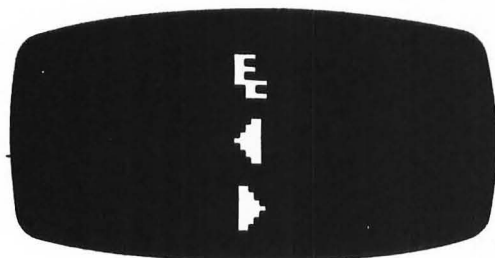
## PSEUDOCODE

Pseudocode may be used as an alternative to flowcharting for program design, the logic that determines the sequence in which processing operations are carried out. In general, this technique uses a code notation which has similarities to BASIC, but is not quite the same.

In pseudocode, we are restricted to just three logic control elements: sequence, selection, and iteration. In pseudocode, the selection element is if . . . then . . . else and the iteration element is perform . . . until. However, the overall structure of the pseudocode need not show all the details of processing, just as a flowchart need not show them.

Sequence requires no special notation. It is simply a convention that unless specified otherwise, operations are carried out from top to bottom in the order written.

# Chapter 14



## ATARI Graphics and Sound

The ATARI 800 allows you to create graphics using BASIC commands and different graphics modes. Using these commands, it is possible to create graphics for games, illustrations, and diagrams. Commands covered in this chapter will include:

GRAPHICS	LOCATE	PUT/GET
COLOR	PLOT	SET/COLOR
DRAWTO	POSITION	XIO

### COMMANDS TO CONTROL THE GRAPHICS

#### Graphics

The graphics command is used to select one of the nine graphics modes. Table 14-1 summarizes the nine modes and the characteristics of each. The graphics command automatically opens the screen, S:(the graphics window), as device #6. So when printing text in the text window, it is not necessary to specify the device code. The mode number must be positive and rounded to the nearest integer. Graphics Mode 0 is a full-screen display while modes 1 through 8 are split screen displays. To override the split-screen, add the characters +16 to the mode number (aexp) in the graphics command. Adding 32 prevents the graphics command from clearing the screen.

To return to Graphics Mode 0 in the direct mode, press the system reset button or type GR.0 and press the return key.

**Graphics Mode 0:** This mode is the 1-color, 2-luminance

**Table 14-1. Modes and Screen Format.**

SCREEN FORMAT						
Gr. Mode	Mode Type	Horiz. (Rows)	Vert. (Col) Split Screen	Vert. (Col) Full Screen	Number Of Colors	RAM Required (Bytes)
0	TEXT	40	-	24	2	993
1	TEXT	20	20	24	5	513
2	TEXT	20	10	12	5	261
3	GRAPHICS	40	20	24	4	273
4	GRAPHICS	80	40	48	2	537
5	GRAPHICS	80	40	48	4	1017
6	GRAPHICS	160	80	96	2	2025
7	GRAPHICS	160	80	96	4	3945
8	GRAPHICS	320	160	192	1/2	7900

(brightness) default mode for the ATARI 800. It contains a 24 by 40 character screen matrix. The default margin settings at 2 and 39 allow 38 characters per line. Margins may be changed by poking LMARGN and RMARGN (82 and 83). Some systems have different margin default settings. The color of the characters is determined by the background color. Only the luminance of the characters can be different. To display characters at a specified location, use one of the following two methods. Note: "sexp" denotes a string expression; "aexp" denotes an arithmetic expression. "Lineno" means line number.

#### Method 1

lineno POSITION aexp1, aexp2    Puts cursor at location  
lineno PRINT sexp                specified by aexp1  
   and aexp2.

#### Method 2

lineno GR.0                        Specifies graphics mode.  
lineno POKE 752,1                Suppresses cursor.  
lineno COLOR ASC(sexp)        Specifies character to be  
   printed.  
lineno PLOT aexpl,                Specifies where to print  
                 aexp2                character.  
lineno GOTO lineno                Start loop to prevent READY  
   from being printed. (GOTO  
   same lineno.)  
   Press BREAK to terminate  
   loop.

Graphics 0 is also used as a clear screen command either in the direct mode or the deferred mode. It terminates any previously

selected graphics mode and returns the screen to the default mode (graphics 0).

**Graphics Modes 1 and 2:** As defined in Table 14-1, these two five-color modes are text modes. However, they are both split-screen modes. Characters printed in Graphics Mode 1 are twice the width of those printed in graphics 0, but are the same height. Characters printed in Graphics Mode 2 are twice the width and height of those in Graphics Mode 0. In the split-screen mode, a print command is used to display characters in either the text window or the graphics window. To print characters in the graphics window, specify device #6 after the PRINT command.

```
Example: 100 GR.1
          110 PRINT #6; "ATARI"
```

The default colors depend on the type of character input. The following table defines the default color and color register used for each type.

**DEFAULT COLORS FOR SPECIFIC INPUT TYPES**

Character Type	Color Register	Default Color
Upper case alphabetical	0	Orange
Lower case alphabetical	1	Light Green
Inverse upper case alphabetical	2	Dark Blue
Inverse lower case alphabetical	3	Red
Numbers	0	Orange
Inverse numbers	2	Dark Blue

Unless otherwise specified, all characters are displayed in upper case noninverse form. To print lower case letters and graphics characters, use a poke 756,226. To return to upper case, use poke 756,224. In Graphics Modes 1 and 2, there is no inverse video, but it is possible to get all the rest of the characters in four different colors.

A split-screen display for Graphics Modes 1 and 2 is shown in Fig. 14-1. The X and Y coordinates start at 0 (upper left of screen). The maximum values are the numbers of rows and columns minus 1 (see Table 14-1).

```
Example: GRAPHICS 1 + 16
```

**Graphics Modes 3, 5, and 7:** These three 4-color graphics

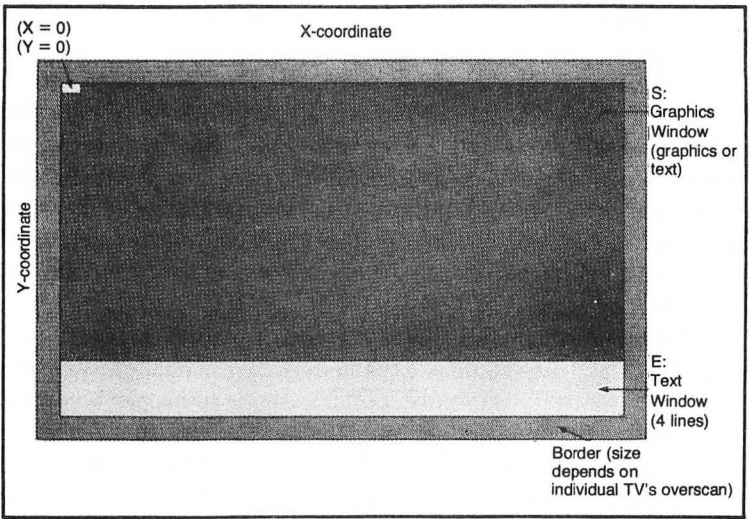


Fig. 14-1. The graphics and text windows.

modes are also split-screen displays in their default state, but may be changed to full screen by adding +16 to the mode number. Modes 3, 5, and 7 are alike except that modes 5 and 7 use more points (pixels) in plotting, drawing, and positioning the cursor. The points are smaller, thereby giving a much higher resolution than that in modes 0, 1, and 2.

**Graphics Modes 4 and 6:** These graphics modes are like modes 5 and 7 respectively except that they use only two colors, and therefore use less memory.

**Graphics Mode 8:** This graphics mode gives the highest resolution of all the modes. As it takes a lot of RAM to obtain this kind of resolution, it can only accommodate a maximum of one color and two different luminances.

**Color (C.):** The value of the expression in the color statement determines the data to be stored in the display memory and used by all subsequent plot and drawto commands until the next color statement is executed. The value must be positive and is usually an integer from 0 through 255. Nonintegers are rounded to the nearest integer. The graphics-display hardware interprets this data in different ways in the different graphics modes. In Text Modes 0 through 2, the number can be from 0 through 255 (8 bits) and determines the character to be displayed and its color.

Tables 14-4 and 14-5 later in this chapter illustrate the internal character set and the character/color assignment.

Graphics Modes 3 through 8 are not text modes, so the data stored in the display RAM simply determines the color of each pixel. Two-color or two-luminance modes require either 0 or 1 (1-bit) and four-color modes require 0,1,2, or 3. The actual color which is displayed depends on the value in the color register which corresponds to the data of 0,1,2, or 3 in the particular graphics mode being used. This may be determined by looking in Table 14-2, which gives the default colors and the corresponding register numbers. Colors may be changed by using the setcolor command described on the following pages.

Note that when BASIC is first powered up, the color data is 0, and when a graphics command (without +32) is executed, all of the pixels are set to 0. Therefore, nothing seems to happen when the plot and drawto commands are used in graphics 3 through 7 when no color statement has been executed. Correct by doing a color 1 first.

**Drawto:** This statement causes a line to be drawn from the last point displayed by a plot to the location specified by aexp1 and aexp2. The first expression represents the X coordinate and the second represents the Y coordinate. The color of the line is the same color as the point displayed by plot.

**Locate:** This command positions the invisible graphics cursor at the specified location in the graphics window, retrieves the data at that pixel, and stores it in the specified arithmetic variable. This gives a number from 0 to 255 for Graphics Modes 0 through 2; 0 or 1 for the 2-color graphics modes; and 0,1,2, or 3 for the 4-color modes. The two arithmetic expressions specify the X and Y coordinates of the point. locate is equivalent to:

POSITION aexp1, aexp2:GET #6, avar

Doing a print after a locate may cause the data in the pixel which was examined to be modified. This problem is avoided by repositioning the cursor and putting the data that was read back into the pixel before doing the PRINT. The following program illustrates the use of the locate command.

```
10 GRAPHICS 3 + 16
20 COLOR 1
30 SETCOLOR 2,10,8
40 PLOT 10,15
50 DRAWTO 15,15
60 LOCATE 12, 15,X
70 PRINT X
```

**Table 14-2. Mode, Setcolor, Color Table.**

Default Colors	Mode or Condition	SETCOLOR (aexp1) Color Register No.	Color (aexp)	DESCRIPTION AND COMMENTS
LIGHT BLUE DARK BLUE BLACK	MODE 0 and ALL TEXT WINDOWS	0 1 2 3 4	COLOR data actually determines character to be plotted	— Character luminance (same color as background) Background — Border
ORANGE LIGHT GREEN DARK BLUE RED BLACK	MODES 1 and 2 (Text Modes)	0 1 2 3 4	COLOR data actually determines character to be plotted	Character Character Character Character Background, Border
ORANGE LIGHT GREEN DARK BLUE BLACK	MODES 3, 5, and 7 (Four-color Modes)	0 1 2 3 4	1 2 3 - 0	Graphics point Graphics point Graphics point — Graphics point (background default), Border
ORANGE  BLACK	MODES 4 and 6 (Two-color Modes) 4	0 1 2 3 0	1 - - - -	Graphics point — — — Graphics point (background default), Border
LIGHT GREEN DARK BLUE BLACK	MODE 8 (1 Color 2 Luminances)	0 1 2 3 4	- 1 0 - -	— Graphics point luminance (same color as background) Graphics point (background default) — Border

On execution, the program prints the data (1) determined by the color statement which was stored in pixel 12, 15.

**Plot:** The plot command is used in Graphics Modes 3 through 8 to display a point in the graphics window. The aexp1 specifies the X-coordinate and the aexp2, the Y-coordinate. The color of the plotted point is determined by the hue and luminance in the color register determined by the last color statement executed. To change this color register, and the color of the plotted point, use setcolor. Points that can be plotted on the screen are dependent on the graphics mode being used. The range of points begins at 1 and extends to one less than the total number of columns (X-coordinate) or rows (Y-coordinate) shown in Table 14-1.

**Position:** The position statement is used to place the invisible graphics-window cursor at a specified location on the screen. This statement can be used in all modes. Note that the cursor does not actually move until an I/O command which involves the screen is issued.

**Put:** In graphics work, put is used to output data to the screen display. This statement works hand-in-hand with the position statement. After a put, the cursor is moved to the next location on the screen. Doing a put to device #6 causes the one-byte input to be displayed at the cursor position. The byte is either an ATASCII code byte for a particular character or the color data.

**Get:** Is used to input the code byte of the character displayed at the cursor position into the specified arithmetic variable. The values used in put and get correspond to the values in the color statement. (Print and input may also be used.)

Doing a print after get from the screen may cause the data in the pixel which was examined to be modified. To avoid this problem, reposition the cursor and put the data that was read, back into the pixel before doing the print.

**Setcolor:** This statement is used to choose the particular hue and luminance to be stored in the specified color register. The parameters of the setcolor statement are defined below:

- aexp1     = Color register (0-4 depending on graphics mode).
- aexp2     = Color hue number (0-15. See Table 14-3).
- aexp3     = Color luminance (must be an even number between 0 and 14; the higher the number, the brighter the display. 14 is almost pure white!)

The ATARI display hardware contains five color registers,



**Table 14-3. Hue (Setcolor Command) Numbers and Colors.**

<b>Colors</b>	<b>Setcolor (aexp2) Numbers</b>
Gray	0
Light orange (gold)	1
Orange	2
Red-orange	3
Pink	4
Purple-blue	6
Blue	7
Blue	8
Light blue	9
Turquoise	10
Green-blue	11
Green	12
Yellow-green	13
Orange-green	14
Light orange	15

*Note: Colors will vary with type and adjustment of TV or monitor used.*

numbered from 0 through 4. The operating system (OS) has five RAM locations (color0 through color4) where it keeps track of the current colors. The setcolor statement is used to change the values in these RAM locations. The setcolor statement requires a value from 0 to 4 to specify a color register. The color statement uses different numbers because it specifies data which only indirectly corresponds to a color register. This can be confusing, so experimentation and study of the various tables in this chapter is recommended.

No setcolor commands are needed if the default set of five colors is used. See Table 14-4. Although 128 different color-luminance combinations are possible, not more than five can be displayed at any one time. The purpose of the color registers and setcolor statement is to specify these five colors.

A program illustrating Graphics Mode 3 and the commands explained so far is shown below:

```

10  GRAPHICS 3
20  SETCOLOR 0,2,8: COLOR 1
30  PLOT 17,1: DRAWTO 17, 10: DRAWTO 9, 18
40  PLOT 19,1: DRAWTO 19, 18
50  PLOT 20,1: DRAWTO 20, 18
60  PLOT 22,1: DRAWTO 22, 10: DRAWTO 30,18
70  POKE 752,1

```

**Table 14-4. Setcolor "Default" Colors.\***

Setcolor (Color Register)	Defaults To Color	Luminance	Actual Color
0	2	8	Orange
1	12	10	Green
2	9	4	Dark blue
3	4	6	Pink or blue
4	0	0	Black

\*"Default" occurs if no setcolor statement is used.  
 Note: Colors may vary depending upon the television monitor type, condition, and adjustment.

```
80 PRINT :PRINT "    ATARI PERSONAL COMPUTERS"
90 GOTO 90
```

The setcolor and color statements set the color of the points to be plotted. The setcolor command loads color register 0 with hue 2 and a luminance of 8. The next 4 lines plot the points to be displayed. Line 90 suppresses the cursor and line 100 prints the string expression ATARI PERSONAL COMPUTERS in the next window. Figure 14-2 shows how this program appears on the screen.

To assign colors to characters in Text Modes 1 and 2, first look up the character number in Table 14-5. Then look at Table 14-6 to get the conversion of that number required to assign a color register to it.

Example: Assign setcolor 0 to lower case "r" in mode 2 whose color is determined by register 0.

1. In Table 14-6, find the column and number for "r" (114-column4).
2. Using Table 14-7, locate column 4. Conversion is the character number minus 32 (114-32=82).
3. Poke the Character Base Address (CHBAS) with 226 to specify lower case letters or special graphics characters; that is, poke 756,226 or CHBAS = poke CHEBAS, 226.  
To return to upper case letters, numbers, and punctuation marks, poke CHBAS with 224.
4. A print statement using the converted number (82) assigns the lower case "r" to setcolor 0 in mode 2.

**XIO:** This special application of the XIO statement fills an area on the screen between plotted points and lines with a non-zero value.

The following steps illustrate the fill process:

1. Plot bottom right corner (point 1).

2. Draw to upper right corner (point 2). This outlines the right edge of the area to be filled.
3. Draw to upper left corner (point 3).
4. Position cursor at lower left corner (point 4).
5. Poke address 765 with the fill color data (1,2, or 3).

This method is used to fill each horizontal line from top to bottom of the specified area. The fill starts at the left and proceeds across the line to the right until it reaches a pixel which contains non-zero data. Fill cannot be used to change an area which has been filled in with a non-zero value so the fill will stop. The fill command will go into an infinite loop if a fill-with-zero command is attempted on a line which has no non-zero pixels. Press the break key or the system reset button to stop the fill if this happens.

The following program creates a shape and fills it with a data (color) of 3. Note that the XIO command draws in the lines on the left and bottom of the figure.

```

10 GRAPHICS 5+16
20 COLOR 3
30 PLOT 70,45
40 DRAWTO 50,10
50 DRAWTO 30,10
60 POSITION 10,45
70 POKE 765,3
80 XIO 18,#6,0,0,"S: "
90 GOTO 90

```

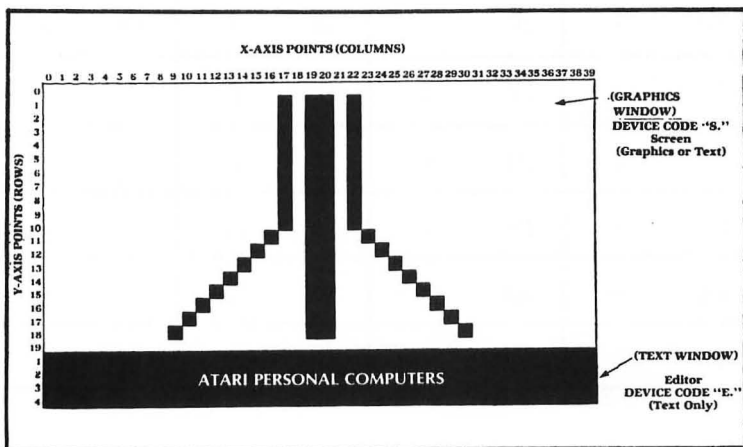


Fig. 14-2. The ATARI symbol as produced by the program.

**Table 14-5. Internal Character Set.**

<b>Column 1</b>				<b>Column 2</b>				
<b>#</b>	<b>CHR</b>	<b>#</b>	<b>CHR</b>	<b>#</b>	<b>CHR</b>	<b>#</b>	<b>CHR</b>	
0	Space	16	0	32	@	48	P	
1	!	17	1	33	A	49	Q	
2	"	18	2	34	B	50	R	
3	#	19	3	35	C	51	S	
4	\$	20	4	36	D	52	T	
5	%	21	5	37	E	53	U	
6	&	22	6	38	F	54	V	
7	'	23	7	39	G	55	W	
8	(	24	8	40	H	56	X	
9	)	25	9	41	I	57	Y	
10	*	26	:	42	J	58	Z	
11	+	27	;	43	K	59	[	
12	,	28	<	44	L	60	\	
13	—	29	=	45	M	61	]	
14	—	30	>	46	N	62	^	
15	/	31	?	47	O	63	—	

Column 3				Column 4			
#	CHR	#	CHR	#	CHR	#	CHR
64		80		96		112	p
65		81		97	a	113	q
66		82		98	b	114	r
67		83		99	c	115	s
68		84		100	d	116	t
69		85		101	e	117	u
70		86		102	f	118	v
71		87		103	g	119	w
72		88		104	h	120	x
73		89		105	i	121	y
74		90		106	j	122	z
75		91	<sup>①</sup>	107	k	123	
76		92		108	l	124	
77		93		109	m	125	<sup>①</sup>
78		94		110	n	126	<sup>①</sup>
79		95		111	o	127	<sup>①</sup>

1. In mode 0 these characters must be preceded with an escape, CHR\$(27), to be printed.

**Table 14-6. Character/Color Assignment.**

		Conversion 1	Conversion 2	Conversion 3	Conversion 4
Mode 0	<sup>2</sup> Setcolor 2	#+32	#+32	#-32	None
		Poke 756,224		Poke 756,226	
Mode 1 Or Mode 2	Setcolor 0	-\$32	#+32	#-32	#-32
	Setcolor 1	None	#+64	#-64	None
	Setcolor 2	#+160	#+160	#+96	#+96
	Setcolor 3	#+128	#+192	#+64	#+128

2. Luminance controlled by Setcolor 1,0, LUM.

**Graphic Control Characters:** These characters are produced when the CTRL key is pressed with the alphabetic keys shown in Fig. 14-3. These characters can be used to draw designs, pictures, and the like in mode 0 and in modes 1, and 2 if CHBAS is changed.

## TEXT MODES CHARACTER PRINT PROGRAM

The following program prints the ATARI characters in their default colors for Text Modes 0,1, and 2. When entering this program, remember that the clear screen symbol “ ” is printed as “}”.

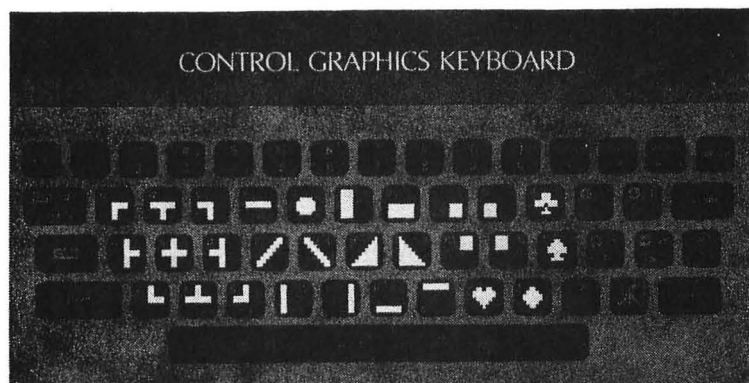
```

1  DIM A$(1)
5  ?“}”:REM CLEAR SCREEN
10 ? “GRAPHICS 0, 1, AND 2 (TEXT MODES)”
20 ? “DEMONSTRATION.”
30 ?“DISPLAYS CHARACTER SETS FOR EACH MODE.”
60 WAIT=1000:REM SUBROUTINE LINE NUMBER
70 CHBAS=756:REM CHARACTER BASE ADDRESS
80 UPPER=224:REM DEFAULT FOR CHBAS
90 LOWER=226:REMLOWER CASE LETTERS — GRAPHICS
95 GOSUB WAIT
100 FOR L=0 TO 2
112 REM USE E: FOR GRAPHICS 0
115 IF L=0 THEN OPEN #1,8,0, “E”:GOTO 118
116 REM USE S: FOR GRAPHICS 0

```

**Table 14-7. Pitch Values for Musical Notes.**

High Notes	C	29
	B	31
	A# or B <sup>b</sup>	33
	A	35
	G# or A <sup>b</sup>	37
	G	40
	F# or G <sup>b</sup>	42
	F	45
	E	47
	D# or E	50
	D	53
	C# or D <sup>b</sup>	57
	C	60
	B	64
	A# or B	68
	A	72
	G# or A <sup>b</sup>	76
	G	81
	F# or G <sup>b</sup>	85
Middle C	F	91
	E	96
	D# or E <sup>b</sup>	102
	D	108
	C# or D <sup>b</sup>	114
	C	121
	B	128
	A# or B <sup>b</sup>	136
	A	144
	G# or A <sup>b</sup>	153
	G	162
	F# G <sup>b</sup>	173
	F	182
Low Notes	D	193
	D# or E <sup>b</sup>	204
	D	217
	C# or D <sup>b</sup>	230
	C	243



**Fig. 14-3. The keyboard graphics characters.**

```

117 OPEN #1, 8, 0, "S:"
118 GRAPHICS L
120 PRINT "GRAPHICS ";L
130 FOR J=0 TO 7:REM 8LINES
140 FOR I=0 TO 31 :REM 32 CHARS/LINE
150 K=32*J+I
155 REM DON'T DISPLAY "CLEAR SCREEN" OR "RE-
    TURN"
160 IF K=ASC("(") OR K=155 THEN 180
165 IF L=0 THEN PUT #1, ASC (" "):REM ESCAPE
170 PUT #1,K:REM DISPLAY CHARS
180 NEXT I
190 PRINT #1;" " :REM END OF LINE
200 IF L<> 2 or J<>3 THEN 240
210 REM SCREEN GULL
220 GOSUB WAIT
230 PRINT #1,"}":REM CLEAR SCREEN
240 NEXT J
250 GOSUB WAIT
265 PRINT "LOW CASE AND GRAPHICS"
270 IF L<>0 THEN POKE CHBAS, LOWER:GOSUB WAIT
275 CLOSE #1
280 NEXT L
300 GRAPHICS 0:END
1000 REM WAIT FOR "RETURN"
1010 PRINT "HIT RETURN TO CONTINUE";
1020 INPUT A$
1030 RETURN

```

## LIGHT SHOW PROGRAM

The following program demonstrates another aspect of ATARI graphics. It uses Graphics Mode 7 for high resolution and the plot and drawto statements to draw the lines. In line 20, the title will be more effective if it is entered in inverse video using the ATARI logo key.

```

10 FOR ST=1 TO 8:GRAPHICS 7
15 POKE 752,1
20 ?:" ATARI's Special Light Show ":SETCOLOR 2,0,0
30 SETCOLOR 1, 2*ST,8:COLOR 2
40 FOR DR=0 TO 80 STEP ST
50 PLOT 0,0: DRAWTO 100,DR

```



```
60 NEXT DR: FOR N=1 TO 800: NEXT N: NEXT ST
70 FOR N=1 TO 2000: NEXT N"GOTO 10
```

## UNITED STATES FLAG PROGRAM

This program involves switching colors to set up the stripes. It uses Graphics Mode 7 plus 16 so that the display uses the full screen. Note the correspondence between the color statements and the setcolor statements. For fun and experimentation purposes, add a sound statement and use a read/data combination to add "The Star Spangled Banner" after line 470. Refer to next section to see how this is done.

```
10 REM DRAW THE UNITED STATES FLAG
20 REM HIGH RESOLUTION 4-COLOR GRAPHICS, NO
   TEXT WINDOW
30 GRAPHICS 7+16
40 REM SETCOLOR 0 CORRESPONDS TO COLOR 1
50 SETCOLOR 0,4,4:RED =1
60 REM SETCOLOR 1 CORRESPONDS TO COLOR 2
70 SETCOLOR 1,0,14: WHITE=2
80 REM SETCOLOR 2 CORRESPONDS TO COLOR 3
90 BLUE =3:REM DEFAULTS TO BLUE
100 REM DRAW 13 RED & WHITE STRIPES
110 C=RED
120 FOR I=0 TO 12
130 COLOR C
140 REM EACH STRIPE HAS SEVERAL HORIZONTAL
   LINES
150 FOR J=0 TO 6
160 PLOT 0,I*7+J
170 DRAWTO 159,I*7+J
180 NEXT J
190 REM SWITCH COLORS
200 C=C+1:IF C> WHITE THEN C=RED
210 NEXT I
300 REM DRAW BLUE RECTANGLE
310 COLOR BLUE
320 FOR I=0 TO 48
330 PLOT 0,I
340 DRAWTO 79,I
350 NEXT I
360 REM DRAW 9ROWS OF WHITE STARS
```

```

370 COLOR WHITE
380 K=0:REM START WITH ROW OF 6 STARS
390 FOR I=0 TO 8
395 Y=4+I*5
400 FOR J=0 TO 4: REM 5 STARS IN A ROW
410 X=K+5+J*14: GOSUB 1000
420 NEXT J
430 IF K<>0 THEN K=0: GOTO 470
440 REM ADD 6TH STAR EVERY OTHER LINE
450 X=5+5*14:GOSUB 1000
460 K=7
470 NEXT I
500 REM IF KEY HIT THEN STOP
510 IF PEEK (764)=255 THEN 410
515 REM OPEN TEXT WINDOW WITHOUT CLEARING
    SCREEN
520 GRAPHICS 7+32
525 REM CHANGE COLORS BACK
530 SETCOLOR 0,4,4: SETCOLOR 1,0,14
550 STOP
1000 REM DRAW 1 STAR CENTERED AT X,Y
1010 PLOT X-1,Y:DRAWTO X+1,Y
1020 PLOT X,Y-1: PLOT X,Y+1
1030 RETURN

```

## VIDEO GRAFFITI

This program requires a joystick controller for each player. Each joystick has one color associated with it. By maneuvering the joystick, different patterns are created on the screen. Note the use of the stick and strig commands.

```

1 GRAPHICS 0
2 ? "VIDEO GRAFFITI"
5 REM X&Y ARRAYS HOLD COORDINATES
6 REM FOR UP TO 4 PLAYERS' POSITIONS.
7 REM COLR ARRAY HOLDS COLORS.
10 DIM A$(1),X(3),Y(3),COLR(3)
128 ? "USE JOYSTICKS TO DRAW PICTURES"
129 ? "PRESS BUTTONS TO CHANGE COLORS"
130 ? "INITIAL COLORS:"
131 ? "JOYSTICK 1 IS RED"
132 ? "JOYSTICK 2 IS WHITE"

```

```

133 ? "JOYSTICK 3 IS BLUE"
134 ? "JOYSTICK 4 IS BLACK (BACKGROUND)"
135 ? "BLACK LOCATION IS INDICATED BY A BRIEF
    FLASH OF RED."
136 ? "IN GRAPHICS 8, JOYSTICKS 1 AND 3 ARE WHITE
    AND 4 IS BLUE."
138 PRINT "HOW MANY PLAYERS (1-4)";
139 INPUT A$:IF LEN(A$)=0 THEN A$="1"
140 JOYMAX=VAL(A$)-1
145 IF JOYMAX<0 OR JOYMAX =4 THEN 138
147 PRINT "GRAPHICS 3 (40X24), 5 (80X48),"
150 PRINT "7 (160X96), OR 8 (320X192)";
152 INPUT A$:IF LEN(A$)=0 THEN A$="3"
153 A=VAL(A$)
154 IF A=3 THEN XMAX=40:YMAX=24:GOTO 159
155 IF A=5 THEN XMAX=80:YMAX=48:GOTO 159
156 IF A=7 THEN XMAX=160:YMAX=96:GOTO 159
157 IF A=8 THEN XMAX=320:YMAX=192:GOTO 159
158 GOTO 147:REM A NOT VALID
159 GRAPHICS A+16
160 FOR I=0 TO JOYMAX:X(I)=XMAX/2+I:Y(I)=YMAX/
    2+I:NEXT I: REM START NEAR CENTER OF SCREEN
161 IF A<>8 THEN 166
162 FOR I=0 TO 2:COLR(I)=1:NEXT I
163 SETCOLOR 1,9,14:REM LT. BLUE
165 GOTO 180
166 FOR I=0 TO 2:COLR(I)=I+1:NEXT I
167 SETCOLOR 0,4,6:REM RED
168 SETCOLOR 1,0,14:REM WHITE
180 COLR(3)=0
295 FOR J=0 TO 3
300 FOR I=0 TO JOYMAX:REM CHECK JOYSTICKS
305 REM CHECK TRIGGER
310 IF STRIG(I) THEN 321
311 IF A<>8 THEN 320
312 COLR(I)=COLR(I)+1:IF COLR(I)=2 THEN COLR(I)=
    0:REM 2-COLOR MODE
313 GOTO 321
320 COLR(I)=COLR(I)+1:IF COLR(I)>=4 THEN COLR(I)=
    0:REM 4-COLOR MODE
321 IF J>0 THEN COLOR COLR(I):GOTO 325
322 IF COLR(I)=0 THEN COLOR 1:GOTO 325

```

```

323 COLOR 0:REM BLINK CURRENT SQUARE ON AND
    OFF
325 PLOT X(I),Y(I)
330 JOYIN=STICK(I):REM READ JOYSTICK
340 IF JOYIN=15 THEN 530:REM NO MOVEMENT
342 COLOR COLR(I):REM MAKE SURE COLOR IS ON
344 PLOT X(I),Y(I)
350 IF JOYIN>=8 THEN 390
360 X(I)=X(I)+1:REM MOVE RIGHT
370 IF X(I)>=XMAX THEN X(I)=0
380 GOTO 430
390 IF JOYIN>=12 THEN 430
400 X(I)=X(I)-1:REM MOVE LEFT
410 IF X(I)<0 THEN X(I)=XMAX-1
430 IF JOYIN<>5 AND JOYIN<>9 AND JOYIN<>13
    THEN 470
440 Y(I)=Y(I)+1:IF Y(I)>=YMAX THEN Y(I)=0:REM MOVE
    DOWN
460 GOTO 500
470 IF JOYIN<>6 AND JOYIN<>10 AND JOYIN<>14
    THEN 500
480 Y(I)=Y(I)-1:IF Y(I)<0 THEN Y(I)=YMAX-1:REM
    MOVE UP
500 PLOT X(I),Y(I)
530 NEXT I
535 NEXT J
540 GO TO 295

```

## USING SOUND

In general, BASIC statements are used to generate musical notes and sounds through the audio system of the television monitor. Up to four different sounds can be “played” simultaneously creating harmony. The sound statement can also be used to simulate explosions, whistles, and other interesting noises.

**Sound:** The sound statement causes the specified note to begin playing as soon as the statement is executed. The note will continue playing until the program encounters another sound statement with the `aexp1` or an end statement. This command can be used in either direct or deferred modes.

The sound parameters are described as follows:

- aexp1= Voice. Can be 0-3, but each voice requires a separate sound statement.
- aexp2= Pitch. Can be any number between 0 and 255. The larger the number, the lower the pitch. Table 14-7 shows the pitch numbers for the various musical notes ranging from two octaves above middle C to one octave below middle C.
- aexp3= Distortion. Can be any even number between 0 and 14. A 0 is used to create a "pure" tone whereas a 12 gives an interesting buzzer sound. A buzzing sound can be produced using two separate sound commands with the distortion value alternating between 0 and 1.
- aexp4= Volume control. Can be between 1 and 15. Using a 1 creates a sound barely audible whereas a 15 is loud. A value of 8 is considered normal. If more than one sound statement is being used, the total volume should not exceed 32 or an unpleasant "clipped" tone will result.

The following example shows how to write a program that will "play" the C scale.

```

10 READ A
20 IF A=256 THEN END
30 SOUND 0,A,10,10
40 FOR W=1 TO 400:NEXT W
50 PRINT A
60 GOTO 10
70 END
80 DATA 29, 31, 35, 40, 45, 47, 53, 60, 64, 72, 81, 91, 96, 108,
    121
90 DATA 128, 144, 162, 182, 193, 217, 243, 256

```

Note that the data statement in line 90 ends with a 256. A glance at Table 14-7 will tell us that this is outside of the designated range. The 256 is used as an end-of-data marker.

### TYPE-A-TUNE PROGRAM

The following program turns the ATARI 800 into a single-scale piano; that is, it assigns musical note values to the keys on the top

row of the computer keyboard. When you are using this program, do not try to play harmony; press only one key at a time.

KEY	MUSICAL VALUE
INSERT	B
CLEAR	B <sup>b</sup> (or A <sup>#</sup> )
0	A
9	A <sup>b</sup> (or G <sup>#</sup> )
8	G
7	F <sup>#</sup> (or G <sup>b</sup> )
6	F
5	E
4	E <sup>b</sup> (or D <sup>#</sup> )
3	D
2	D <sup>b</sup> (or C <sup>#</sup> )
1	C

```

10 DIM CHORD(37),TUNE(12)
20 GRAPHICS 0:?:? "          TYPE-A-TUNE PROGRAM"
25 ? :? "PRESS KEYS 1-9,0,<,> TO PRODUCE NOTES.";
27 ? "RELEASE ONE KEY BEFORE PRESSING THE
   NEXT."
28 ? "OTHERWISE THERE MAY BE A DELAY."
30 FOR X=1 TO 37: READ A:CHORD(X)=A:NEXT X
40 FOR X=1 TO 12:READ A:TUNE(X)=A:NEXT X
50 OPEN #1.4.0,"K:"
55 OLDCHR=-1
60 A=PEEK(764):IF A=255 THEN 60
63 IF A=OLDCHR THEN 100
65 OLDCHR=A
70 FOR X=1 TO 12:IF TUNE(X)=A THEN SOUND 0,
   CHORD(X),10,8:GOTO 100
80 NEXT X
100 I=INT(PEEK(53775)/4):IF (I/2)=INT(I/2) THEN 60
110 POKE 764,255:SOUND 0,0,0,0:OLDCHR=-1: GOTO 60
200 DATA 243,230,217,204,193,182,173,162,153,144,136,
   128,121,114,108,102,96,91,85,81,76,72,68,64,60
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

```

To play "Twinkle, Twinkle Little Star" press the following keys:

1,1,8,8,0,0,8	6,6,5,5,3,3,1
8,8,6,6,5,5,3	8,8,6,6,5,5,3
1,1,8,8,0,0,8	6,6,5,5,3,3,1

To play the “Marine’s Hymn” press the following keys:

1,5,8,8,8,8,1,8  
1,5,8,8,6,3,1

The following keys are pressed to play “London Bridge Is Falling Down”:

8,9,8,6,5,6,8	3,5,6	5,6,8
8,9,8,6,5,6,8	3,8,5,1	

## COMPUTER BLUES

This program generates random musical notes to “write” some very interesting melodies for the programmed bass.

```

1  GRAPHICS 0:?:? “          COMPUTER BLUES”:?
2  PTR=1
3  THNOT=1
5  CHORD=1
6  PRINT “BASS TEMPO (1=FAST)”;
7  INPUT TEMPO
8  GRAPHICS 2+16:GOSUB 2000
10 DIM BASE(3,4)
20 DIM LOW(3)
25 DIM LINE(16)
26 DIM JAM(3,7)
30 FOR X=1 TO 3
40 FOR Y=1 TO 4
50 READ A:BASE(X,Y)=A
60 NEXT Y
70 NEXT X
80 FOR X=1 TO 3:READ A:LOW(X)=A
90 NEXT X
95 FOR X=1 TO 16:READ A:LINE(X)=A:NEXT X
96 FOR X=1 TO 3
97 FOR Y=1 TO 7
98 READ A:JAM(X,Y)=A:NEXT Y:NEXT X
100 GOSUB 500

```

```

110 T=T+1
115 GOSUB 200
120 GOTO 100
200 REM PROCESS HIGH STUFF
205 IF RND(0)<0.25 THEN RETURN
210 IF RND(0)<0.5 THEN 250
220 NT=NT+1
230 IF NT>7 THEN NT=7
240 GOTO 260
250 NT=NT-1
255 IF NT<1 THEN NT=1
260 SOUND 2,JAM(CHORD,NT),10,NT*2
280 RETURN
500 REM PROCESS BASE STUFF
510 IF BASS=1 THEN 700
520 BDUR=BDUR+1
530 IF BDUR<>TEMPO THEN 535
531 BASS=1:BDUR=0
535 SOUND 0,LOW(CHORD),10,4
540 SOUND 1,BASE(CHORD),THNOT),10,4
550 RETURN
700 SOUND 0,0,0,0
710 SOUND 1,0,0,0
720 BDUR=BDUR+1
730 IF BDUR<>1 THEN 800
740 BDUR=0:BASS=0
750 THNOT=THNOT+1
760 IF THNOT<>5 THEN 800
765 THNOT=1
770 PTR=PTR+1
780 IF PTR=17 THEN PTR=1
790 CHORD=LINE(PTR)
800 RETURN
1000 DATA 162,144,136,144,121,108,102,108,108,96,91,96
1010 DATA 243,182,162
1020 DATA 1,1,1,1,2,2,2,2,1,1,1,1,3,2,1,1
1030 DATA 60,50,47,42,40,33,29
1040 DATA 60,50,45,42,40,33,29
1050 DATA 81,68,64,57,53,45,40
2000 PRINT #6:PRINT #6:PRINT #6
2005 PRINT #6;"          Computer"

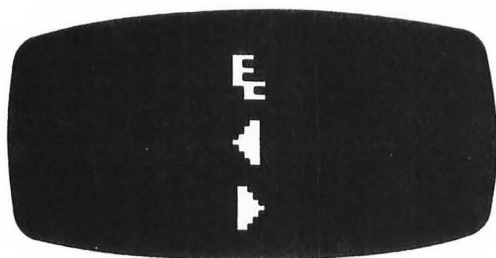
```



2006 PRINT #6  
2010 PRINT #6;“  
2030 RETURN

Blues”

## Chapter 15



### Programming in Machine Language

Machine language is the simplest form of computer language (for the computer that is). It consists of the binary instruction codes that define the basic computer instruction. Each instruction is defined by a unique binary code. When that code is interpreted by the computer, the CPU and other sections of the computer carry out those specific operations defined by that instruction.

Programs can be written in machine language by listing the instructions in the proper sequence. Such a sequence of instructions comprises the program that results in some worthwhile end result.

Machine language is by far the most difficult of all computer languages to use. It not only requires a knowledge of the computer's architecture and operation, but it is tedious and time-consuming. Machine language programming, using binary numbers is also error prone, and is therefore sometimes simplified by using octal or hexadecimal codes instead of binary codes. Despite this improvement, machine language programming is difficult. It is used for writing only short, simple programs. When longer and more complex programs are to be written, machine language programming is abandoned in favor of assembly-language programming.

The ATARI 800 contains a 6502 microprocessor and it is possible to call 6502 machine code subroutines from BASIC using the USR function. Short routines may then be entered into a program by hand assembly. Before the ATARI 800 returns to BASIC, the assembly language routine must do a pull accumulator (PLA)

instruction to remove the number (N) of input arguments off the stack. If this number is not 0, all of the input arguments must be popped off the stack also using PLA.

The subroutine should end by placing the low byte of its result in location 212 (decimal). It should 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.

The ADR (address) function 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 of strings, and then use this address as one of the USR input arguments.

We will take a look at an actual program very soon, but before we do, let's review assembly language. Assembly-language programming is the next step up from machine language. It still uses the individual computer instructions for writing programs, but instead of referring to these instructions by binary, octal, or hexadecimal codes, each instruction is referred to by a shorthand term. Each instruction is referred to by a short multiletter designator known as a mnemonic. For example, the instruction that causes the addition of binary numbers is referred to by the mnemonic ADD. The mnemonic for a store accumulator instruction might be STA. These mnemonics eliminate the need to deal with binary code. In addition, assembly-language programming provides the ability to identify memory locations by symbolic names rather than by actual address numbers. For example, the memory location designated to hold the sum of an arithmetic operation might simply be referred to as SUM. The actual memory address of that memory location need not be used.

When you use assembly language, you will usually also use a program called an assembler. The assembler is a translator program that resides in memory and converts your program, written in mnemonics and symbolic addresses, into machine code which can be executed by the computer. The assembler looks at each mnemonic operation code and symbolic address and converts it into its appropriate binary operation code or address. Each mnemonic typically results in the creation of one binary instruction.

Assembly language programming greatly speeds up and simplifies the programming process. It eliminates many of the complications of dealing with binary numbers. Yet because assembly language programs are capable of using the individual computer

instructions, powerful and efficient programs can be written.

The following program, Hexcode Loader, provides the means of entering hexadecimal codes, converting each hexadecimal number to decimal, and storing the decimal number in an array. The array is then executed as an assembly-language subroutine. To use this program, first enter it. Then save it on disk or cassette for future use.

```
10  GRAPHICS 0:PRINT "HEXCODE LOADER PRO-
    PROGRAM":PRINT
20  REM STORES DECIMAL EQUIVALENTS IN ARRAY A,
    OUTPUTS IN PRINTED 'DATA STATEMENTS' AT
21  REM LINE NUMBER 1500
30  REM USER THEN PLACES CURSOR ON PRINTED
    OUTPUT LINE, HITS "RETURN", AND ENTERS
31  REM REST OF BASIC PROGRAM INCLUDING USR
    STATEMENT.
40  DIM A(50),HEX$(5)
50  REM INPUT, CONVERSION, STORAGE OF DATA.
60  N=0:PRINT"ENTER 1 HEX CODE. IF THERE ARE NO
    MORE, ENTER 'DONE'.";
70  INPUT HEX$
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 60
190  REM PRINT OUT DATA LINE AT 1500
200  GRAPHICS 0: PRINT "1500 DATA";
210  C=0
220  C=C+1
230  IF A(C)=999 THEN PRINT "999":STOP
240  PRINT A(C);",";
250  A(C)=0
260  GOTO 220
300  PRINT "PUT CORRECT NUMBER OF HEX BYTES IN
    LINE 1000.":STOP :REM TRAP LINE
```

```

999  REM ** EXECUTION MODULE **
1000 CLR :BYTES=0
1010 TRAP 300:DIM E$ (1),E(INT(BYTES/6)+1)
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

```

Once this much of the program has been typed, add your own BASIC language part of the program. Start at line 1080 and include the USR function that calls the machine-language subroutine. Some sample programs follow.

Count the total number of hex codes to be entered and enter this number on line 1000 when requested. If another number is already entered, simply replace it.

Run the program and enter the hexadecimal codes of the machine-level subroutine. Press the return key after each entry. After the last entry, type done and press the return key.

Now the data line (1500) displays on the screen. It will not be entered into the program until the cursor is moved to it and the return key is pressed.

Add a program line 5 goto 1000 to bypass the hexcode loader (or delete the hexcode loader through line 260). Now save the completed program by using csave or save. It is important to save the program before executing the part of the program containing the USR call. A mistake in a machine-language routine may cause the system to crash. If the system does hang up, press the system reset button. If the system doesn't respond, turn power off and on again, reload the program, and correct the mistake.

The following sample programs can each be entered into the Hexcode Loader program. The first program prints NOTHING IS MOVING while the machine-language program changes the colors. The second simple program displays a BASIC graphics design and then changes the colors.

```

1080 GRAPHICS 1+16
1090 FOR I=1 TO 6
1100 PRINT #6; "nothing is moving!"
1110 PRINT #6; "NOTHING IS MOVING!"
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:NEXTI:GOTO 1150

```

After entering this program, change line 1000 to read:

```

1000 CLR:BYTES = 21

```

Type RUN and press the return key.

Now enter the hexadecimal codes column by column as shown.

68	2	
A2	E8	
0	E0	
AC	3	
C4	90	
2	F5	
BD	8C	
C5	C7	
2	2	
9D	60	
C4		BYTES = 21

When you have entered them all, type DONE and press the return key. Now place the cursor after the last entry (999) on the data line and press the return key.

Now run the program by typing GOTO 1000 and pressing the return key, or if line 5 has been added, type RUN and press the return key. Press the break key to stop the program and delete line 5.

The second program, which follows, should be entered in place of the NOTHING IS MOVING program. Be sure to check the BYTES = \_\_\_\_\_ count in line 1000.

```

1080 GRAPHICS 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

```

Address	Object Code	Line No.	Label	Mnemonic	Data
02C4		0100			Routine to rotate COLOR data From one register to another. 4 colors are rotated.
02C5		0110			
02C6		0120			
02C7		0130			
		0140			Operating system address
		0150			Color 0 = \$02C4
		0160			Color 1 = \$02C5
		0170			Color 2 = \$02C6
		0175			Color 3 = \$02C7
		0180			Machine program starting address*
6000	68	0190		* =	\$6000
6001	A200	0200		PLA	
6003	ACC402	0210		LDX	#0
6006	BDC502	0220		LDY	Color0
6009	9DC402	0230	Loop	LDA	Color1,X
600C	E8	0240		STA	Color0,X
600D	E002	0250		INX	
		0260		CPX	#3
600F	90F5	0270		BCB	Loop
6011	8CC602	0280		STY	Color3
6014	60	0290		RTS	
Assembler Prints This		This Portion is Source Information Programmer Enters Using ATARI Assembler Cartridge			

**Table 15-1. Assembler Routine to Rotate Colors.**

# Indicates data (source)  
 \* Routine is relocatable  
 \$ Indicates a hexadecimal number

```

1190 NEXT X
1200 X=USR(ADR(E$)+1)
1210 FOR I=1 TO 15:NEXT I
1220 GOTO 1200

```

Type RUN and press the return key.

Enter the hexadecimal codes for this program column by column.

68	2
A2	E8
0	EO
AC	2
C4	90
2	F5
BD	8C
C5	C6
2	2
9D	60
C4	BYTES = 21

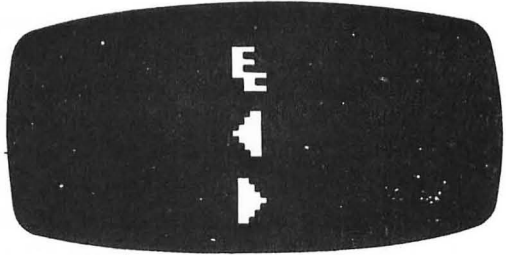
When you have entered them all, type DONE and press the return key. Now place the cursor after the last entry (999) on the data line and press the return key.

Now run the program by typing GOTO 1000 and pressing the return key, or add line 5 GOTO 1000, type RUN and press the return key. Press the break key to stop program and delete line 5.

Table 15-1 illustrates an assembler subroutine used to rotate colors which might prove useful. It is included here for your information.



## Chapter 16



### Continuing Education

Computer technology isn't something you can learn quickly by simply buying a microcomputer and studying a couple of books. Not that it isn't possible to teach yourself this way, but you'll save a lot of time if you start with some professional instruction.

Ideally, the best way to learn computer technology is to attend one of the bona fide schools offering courses in computer science, but for some of you, attending one of these colleges or schools may not always be possible. You have to earn a living and, therefore, full-time instruction is out of the question. You may find that a home-study school can solve the problem. Location, working hours, age, and educational background are not barriers for the serious, highly-motivated home-study student. Furthermore, a person taking a home-study course can progress at his or her own pace—as fast as he or she can master the lessons or as slowly and irregularly (within reason) as necessary. Two correspondence schools offering courses in computer technology include:

NRI

McGraw-Hill Continuing Education Center  
3939 Wisconsin Avenue  
Washington, DC 20016

and

National Technical Schools  
4000 South Figueroa Street  
Los Angeles, CA 90037

NRI's advanced course of study called the Professional Course is designed to get qualified individuals into the exciting and rapidly growing field of computer technology as quickly as possible. The course was created by selecting the lessons from NRI's Master Course that deal specifically with computers. The Professional Course lesson texts are described in detail later on. The first lesson in the course is "Introduction to Computers." The final lesson is "Digital Troubleshooting Equipment."

The Professional Course lesson texts are supplemented with thorough training on the Radio Shack TRS-80 model III microcomputer. This training is accomplished by interfacing the NRI Discovery Lab with the TRS-80. The lab is the only equipment that is not included in the tuition price of the course. If you've previously taken an NRI electronics course perhaps you have a Discovery Lab already. If not, you can order the Discovery Lab through NRI for \$100.

The TRS-80 model III with 16K RAM is included in their Professional Course training. This Radio Shack microcomputer comes fully assembled and ready to use. It has many practical applications that you can utilize after you've finished using it as a training device. Training on the TRS-80 will give you the hands-on experience you will need to move ahead in the world of microcomputer technology.

The NRI Professional Course in Microcomputer training prepares you for the tremendous employment surge in the field of computer service technicians—a growth that has soared from 63,000 back in 1978 and will reach 160,000 in 1990. This phenomenal growth predicted by the U.S. Department of Labor, Bureau of Labor Statistics will occur in an incredibly short period—154% growth in 12 years!

In the fields of computers and electronics, education is not only necessary to get a job but also important to advance in your career. NRI courses provide a sound technical education as a foundation for entry level jobs at the technician level. In addition, NRI courses are ideal for continuing education, the on-going process of refreshing your fundamentals and gaining new knowledge of high technology subjects. Continuing education will keep you competent and provide the kind of knowledge you will need to advance on the job or land a new, better job. If you are serious about a technical career, an NRI course can get you into one sooner and keep you going longer.

An NRI course can open many opportunities for earning extra income in your spare time. Many NRI graduates have opened part-time service shops in their homes to make extra money. That opportunity and many others still exists today. Completion of an NRI course could help you get an FCC license that would allow you to do part-time service and installation work on mobile, marine, aircraft, or CB radios. There is also a great need for part-time technicians to repair microcomputers. Part-time and "moonlighting" opportunities abound in electronics, and an NRI course will let you take advantage of them. Part-time work will often open up unexpected opportunities that blossom into a full-time business or a rewarding career.

Electronics and computers are great hobbies and leisure-time activities. There are literally thousands of individuals that enjoy electronics as a general-interest, spare-time diversion. Electronics is fun, interesting, challenging, and exciting. It involves a variety of interests including amateur radio, CB, personal computing, hi-fi/audio, model trains, radio control, and just plain experimenting, kit building and the like. An NRI course can get you into a hobby quickly and easily or enhance and enrich your present electronic hobby. After all, a major part of any hobby is learning more about a subject in your spare time. What better way than with a home-study course such as those offered by NRI?

## **NATIONAL TECHNICAL SCHOOL**

The subjects covered in NTS's microcomputer course are very similar to those offered by NRI, except that NTS utilizes the Heath 89 computer instead of the Radio Shack TRS-80. In fact, when sample lessons supplied by both schools were examined, each set of lessons showed strong and weak points which were balanced-out by similar inconsistencies in the lessons from the other school. Both schools are approved by the Veterans Administration for GI training, and both have a staff of well-known professional consultants.

Courses for both schools are surprisingly complete, well thought out, and well written. All the material was very easy to read and understand, and the many illustrations created interest throughout. Any person who finishes either of these courses and who obtains a complete reference library (and knows how to use it) should have the basic knowledge necessary to become a good employee for any computer firm. The trained hobbyist will be able to accomplish much more on his home computer than he or she would otherwise.

## TYPICAL CORRESPONDENCE LESSONS

The following is a sampling of the lessons and kits found in typical correspondence courses. Of course, the exact contents will vary from school to school, but these examples should give the reader a good idea of what is in store for him or her.

**Introduction to Electronics:** You get off to a fast start with an overview of the microcomputer industry. You get right to the heart of the business with the study of what electricity is, how electrical current is made to flow in a circuit, and the relationship between current, voltage, and resistance.

**How Electricity is Produced for Electronics:** You explore the important types of batteries such as dry cells, mercury cells, manganese cells, lead-acid cells and nickle-cadmium cells. You are then introduced to direct current (dc) and to alternating current (ac) generators, and you carefully analyze how they work.

**Current, Voltage, and Resistance:** You are now ready for a close-up examination of current, voltage, and resistance, discovering how units of each are measured and how voltage sources act when connected. You also learn how resistance limits the flow of current. Since no discussion of electricity is complete without carefully considering Ohm's Law (which is one of the most important laws of electronics), it is presented here in a readily understandable manner. You quickly learn how to use it to determine circuit parameters.

**Series and Parallel Circuits:** In this lesson you get into more complex circuits, learning the difference between series and parallel circuits. You learn all about resistance in series circuits, voltage drops, and the very important relationship between the voltage drops in a series circuit and the source voltage. Building on this knowledge, you learn how to find the resistance in a parallel circuit. The lesson concludes with a study of voltage and current in parallel circuits.

**How Resistors are Used:** Now that you have the basic background to understand resistors and how they are used in circuits, you study the watt, the wattage rating of resistors, and the transfer of power. You learn about resistor values, the color code used to indicate the value of a resistor, and about resistors with special characteristics, such as the temperature-sensitive thermistor and the voltage-dependent varistor.

**How Coils Are Used:** You discover the different types of coils, their uses and their basic function in electronic circuits. You learn about magnetic circuits and compare them to circuits you have

already covered. You learn about inductive reactance and the opposition that a coil offers to alternating current. You study Lenz's Law for coils and learn how changing flux linkages can produce a voltage. You study Kirchhoff's Voltage Law, and learn how Ohm's Law is applied to simple circuits having resistance and inductance.

**How Capacitors Are Used:** In this lesson you study capacitors (once called condensers) and examine the differences between the various types (variable, paper, mica, ceramic and electrolytic). You learn how these capacitors are made, and how they are used. You learn how a capacitor stores electricity, and how it works in ac circuits. You also learn about capacitive reactance.

**How Resistors, Coils, and Capacitors Are Used Together:** Now that you understand resistance, capacitance, and inductance, you see how circuits which contain all three can form resonant circuits. You learn about both series-resonant and parallel-resonant circuits. You learn how important resonant circuits are and how they are used in electronics.

**How Transistors Work:** You begin your study of the theory behind transistors. You learn what a semiconductor is and how semiconductor materials are combined to form diodes and transistors. You learn about the junction diode, the Zener diode, the tunnel diode, the pin diode, and the varactor diode. You study npn transistors and pnp transistors and learn how field-effect transistors work. By the time you have completed this lesson, you will know what a MOSFET is, and how it works.

**How Transistors Are Used:** You learn about the three principal transistor circuits: the common-base circuit, the common-emitter circuit, and the common-collector circuit. You learn how these circuits are used in amplifiers. You learn how the field-effect transistor is used as an amplifier and how the gain of a dual-gate field-effect transistor can be controlled.

**Integrated Circuits:** Here is an invaluable lesson showing how transistors can be integrated into entire circuits and used in computers and other equipment. This lesson is rounded out with a study of both linear and digital integrated circuits.

**Printed Circuit Boards:** You get your first exposure to how printed circuit boards are used to provide a convenient, low-cost, and reliable means of mounting and interconnecting electronic components. You learn how these boards are made and what materials are used. You also learn how to lay out typical printed circuit boards and how to repair them.

**Periodic Waves and Time Constants:** This lesson pro-

vides an in-depth study of the types of signals and circuits used in control systems, computers, and other advanced electronic systems. You study periodic waveforms and pulses as well as the RC and RL circuits which are used to shape the pulses and other waveforms.

**Cathode Ray Oscilloscopes:** This lesson explains in detail how the cathode-ray tube is combined with various circuits to form a complete cathode-ray oscilloscope. You see how the oscilloscope is used as a tool to observe and measure waveforms in circuits.

**Relays and Relay Circuits:** Now you analyze the construction and operation of various types of relays. You see how they perform remote switching operations and are able to control large amounts of power under the control and direction of small signals.

**Regulated Power Supplies:** You explore the basic half-wave and full-wave rectifier circuits and elementary filter circuits. You even get a full explanation of how active regulators are used to provide pure dc for electronic circuits. You also become familiar with the basic series and shunt regulators, as well as with the special switching and other feedback regulators.

**Transistor Amplifier Circuits:** In this lesson you study common transistor amplifier circuits in detail and see how various configurations affect input and output impedance, voltage and power gain, and phase shift. You also consider the differential amplifiers and special compensating techniques used for stabilization.

**How Oscillators Work:** Now you can concentrate on how transistors and integrated circuits can be used with LC circuits, RC circuits, and crystals to produce oscillation. You study practical circuits that generate the sinusoidal and pulse waveforms useful in rf and computer timing applications. You investigate the operation of several types of special digital circuits including the monostable multivibrator and the astable multivibrator.

**Introduction to Computers:** You are now ready for a close-up look at computers and their applications. You learn the basic structure of computers, with special emphasis on the different types of computers and their various circuits.

**How Computers Are Used:** The entire thrust of this lesson is a well organized overview of how computers are used. A wide variety of computer applications are discussed, paying particular attention to their business and scientific applications. You explore special topics such as simulation, time sharing, and process control.

**Basic Computer Arithmetic:** This lesson provides a review of the decimal number system. It then teaches you what binary

numbers are and how they relate to decimal numbers. You go on to study the forms used for binary addition, subtractions, multiplication and division in whole and fractional number applications. When you complete this lesson you'll see the relationships between binary, octal, hexadecimal, and decimal numbers clearly.

**Digital Codes and Computer Arithmetic:** Now you begin a close examination of fixed and floating point numbers, and the methods used by computers to handle these numbers. You investigate special codes such as the Gray code and the BCD code. You learn about the complement representation of negative numbers in binary and the popular ASCII code used by computers.

**Digital Logic Circuits:** Here you observe the basic transistor inverter used as a switch to represent the two binary states, one and zero. Basic, AND, and OR gates are covered using simple diode logic. And, you learn all about the more complex logic elements such as NAND, NOR, and EXCLUSIVE OR using RTL, DTL, TTL, ECL, MOS, and CMOS logic elements. The characteristics of the various logic families are discussed clearly and simply.

**Boolean Algebra and Digital Logic:** This lesson contains an introduction to the special Boolean math used to analyze, design, and optimize combinational logic circuits. This is the math that permits complex logic networks to be expressed in easily manipulated equations which can be translated directly into optimized circuits using simple logic gates.

**Flip-Flops, Registers, and Counters:** You now look at the various types of flip-flop circuits used to store and manipulate binary data. Here you learn that special registers are made up of combinations of flip-flops to store and shift large numbers of binary digits. You also see how other combinations of special flip-flops form binary and BCD counter or divider circuits.

**How Digital Logic Is Used:** You gain an understanding of how easily any complex logic function can be performed using only basic NAND and NOR gates. The most widely used combinational logic circuits are presented in such detail that you'll have no trouble understanding them. They include encoders, multiplexers, demultiplexers, comparators, and parity generators.

**Computer Arithmetic Operations:** You learn how the arithmetic section of this computer uses logic to perform its functions and operations on fixed and floating point numbers. You become familiar with methods that are used to handle negative numbers and you discuss hardware versus software handling of the multiply and divide operations.

**How Digital Computers Operate:** You study the major elements of the digital computer and discover how each section depends on the others for proper operation and execution of stored programs.

**Register Transfer and Addressing:** You learn the importance of registers in the computer. Special registers such as accumulators, index registers, program counters, and data registers are discussed in detail. You find out how memory as well as input and output can be considered to be registers when you are dealing with the flow of data through the computer.

**Computer Memories:** Here you are introduced to the various types of memories used in computers. You consider volatile as well as nonvolatile memory, paying particular attention to semiconductor memory and magnetic memory such as tape and disk.

**Computer Input/Output:** You learn how the I/O section of a computer operates. In this lesson, special emphasis is placed on becoming acquainted with some of the more popular types of peripheral equipment used with computers.

**Computer Peripheral Equipment:** Peripheral equipment makes up a large portion of any computer system so it is important for you to learn how the different types of devices operate. This lesson carefully surveys printers, magnetic tapes, floppy disks, and terminals.

**Data Conversion Systems:** Analog-to-digital (A/D) and digital-to-analog (D/A) conversion techniques are covered in detail. Here you also get into related topics such as multiplexing and sample-and-hold amplifiers. You round out this lesson by considering how these units are combined with computers to form various types of data conversion and data acquisition systems.

**Microprocessors and Microcomputers:** The computer-on-a-chip is the most powerful kind of electronic component. Called the microprocessor, these devices are used in complete systems called microcomputers. The elements that make up microcomputers of all kinds are described, and examples of some popular microprocessors are shown. The ways that microprocessors are combined with memory and input/output chips to create complete computers for business and industrial applications are clearly and concisely illustrated.

**Microcomputer Applications:** Already, millions of microprocessors and microcomputers have been installed in consumer products, industrial-process control systems, and small businesses. In this lesson, several examples of popular applications are



illustrated. Examples include: (a) one-chip microcomputers in high-volume consumer appliances; (b) single-board microcomputers used in industrial control; (c) special-purpose computers developed for industrial applications like automatic bowling scoring systems and telephone PABX's; and for small business computer uses, including payroll, inventory control, and project scheduling.

**Microprocessors and Support Circuits:** To adequately understand the operation of a microprocessor, you must be thoroughly familiar with the electrical characteristics of major products and the support chips required to make them work in practical circuits. Support circuits covered here include clock generators, bus buffers, and status latches. Also pinpointed are the differences between synchronous and asynchronous storage reference systems, and the means by which these systems are supported by additional ICs.

**Memories and Input/Output Ports:** The structure of memory systems and input/output ports are amazingly similar, and are treated together. You learn the various ways the different memory technologies are used. You also study the support circuitry required to decode and select memory chips for both eight and sixteen-bit word systems. You discover that some of the popular methods of implementing simple serial and parallel I/O ports are really special cases of memory architectures, and that the same decoding methods apply.

**Interface Circuits:** Although input/output ports are essential to a microcomputer, most applications require specialized ways of interfacing to unique sensors and actuators. In this lesson you concentrate on those additional circuits and see how they work. You explore the general rules that all interfaces follow and study some specific examples of interfaces. Included in these examples are audio cassette interfaces and simple analog interfaces for driving conventional meters.

**Register Transfers in Software:** Now you are ready to learn the simplest approach to programming any microcomputer. This technique depends upon understanding all of the registers that exist, and transferring data among these registers to achieve some objective. You learn the techniques used for analyzing a computer's instruction set and register complement. You also get detailed step-by-step instructions for writing simple programs that use that instruction set effectively.

**Software and Programming:** The important subject of programming is introduced here. You get a general overview of computer software including types of programming languages,

machine language, assembly language, higher-level languages, higher-level compiler languages such as FORTRAN, and interpreters such as BASIC, and programming and software systems.

**Higher Level Languages:** This lesson discusses the need for the higher level languages and how they are used by computing machines. Many popular languages are covered. A detailed explanation, examples, and practice problems are included for the BASIC and FORTRAN languages.

**Programming Languages:** Although programming can be done in binary and assembly forms, the use of higher-level programming languages is more popular and productive. In this lesson you study the major features of the BASIC language, as well as some features of the FORTRAN, PASCAL, and COBOL languages. Also covered are other languages like FORTH and PL/M.

**Development Environments:** You begin a careful analysis of the various ways to develop microcomputer hardware and software, including simple kits, software development systems, and complete multiterminal systems. You discover the differences between cross-support and native-support products, and the need for compatible program storage media in the cross-support environment. You also become familiar with various software tools, including assemblers, compilers, editors, interpreters and operating systems.

**Digital Troubleshooting Equipment:** This lesson carefully surveys the basic tools used with microcomputers that are different from the more conventional electronics instrumentation. You discover the advantages of using equipment such as the digital delayed-sweep scope. You also learn how to use such special troubleshooting equipment as portable front panels (such as the Intel 820 Scope), logic state analyzers, and in-circuit emulation.

After studying topics like those described in this chapter, you'll be one of the new breed of computer technician familiar not only with programming and operations, but with the mechanical and electronic nature of the expanding world of computers.

The following describes the various kits that currently accompany correspondence courses in microcomputers and microprocessors. The examples are specifically from NRI's courses.

**Kit No. 1:** In your first training kit, you begin your study of actual electronic circuits, the ones you need to know about to be the new breed of computer specialist; the parts you need are included.

Even if you don't know one end of a soldering iron from the other, you can use this kit. The instructions in your first training kit

take you step-by-step through ten carefully-written experiments which develop your basic electronics skills. You learn how to identify and install parts and make good solder connections to both component terminals and printed circuit boards. It isn't long before you recognize symbols and know how to construct circuits using schematic diagrams. These fundamentals are your foundation for sound troubleshooting and repair techniques.

Using your batteries, resistors, and the LED (light-emitting diode) supplied as part of your kit, you work on actual circuits. You see the effect of changing the resistance or voltage in the circuit as you study both series and parallel circuits. Using NRI's new Action Audio cassettes and specially coordinated diagrams, you learn the operation and practical applications of your Beckman 3½ digit digital multimeter. You learn the professional way to take voltage, current, and resistance measurements. The DMM becomes an integral part of your training when you use it in your experiments to demonstrate the basic fundamentals of electronics. You'll use it later to measure power supply voltage and to verify the performance of discrete components such as diodes, resistors, and transistors found on circuit boards in computers.

## **Specifications**

Display:

3½ digit liquid crystal display (LCD) with a maximum reading of 1999.

Power:

Single, standard 9-volt transistor battery.

Maximum Voltage:

1500 Vdc or peak ac.

Dimensions:

6.85" long × 3.65" wide × 1.8" high.

Weight:

16 ounces including battery.

Case:

High-impact ABS plastic, recessed switch and display.

Dc Volts:

100V to 1000V in 5 ranges.

Ac Volts:

100V to 1000V in 5 ranges.

Dc Current:

100 nA to 2A in 5 ranges.

Ac Current:

100 nA to 2A in 5 ranges.

Resistance:

0.1 to 20 M in 6 ranges.

**Kit No. 2:** You receive as part of your second kit, a professional hand-held digital multimeter. It's the basic, indispensable tool for all computer specialists. You'll use this precision instrument for all voltage, resistance and current measurements in the experiments you perform in this kit, and you'll find it invaluable after you've completed your course. You'll use it then to measure voltages of power supplies and to verify the performance of discrete components such as diodes, resistors, and transistors found on most circuit boards in computers.

Using new audio cassettes and carefully coordinated diagrams and schematics, you're personally talked through the operation and practical applications of your digital multimeter (DMM). By following the step-by-step instructions on the cassette, you learn why it is important to establish a reference point when taking measurements and that a point in the circuit can be both positive and negative.

You learn about voltage dividers and discover how a shunt resistor, called a bleeder, can reduce variations in the output voltage. You also learn how to check continuity by making voltage measurements with your DMM. As you listen to the carefully-sequenced instructions, you put the ohmmeter section of your DMM into operation, learning how to trace circuits and measure series and parallel resistor combinations.

Because you are working with NRI's Action Audio cassette, your hands are free to perform the necessary operations and you can make your visual observations at the same time. You avoid the necessity of looking back and forth between printed instructions and your digital multimeter, thus saving time and reinforcing your learning.

**Kit No. 3:** This kit introduces you to ac circuits and the particular components used in them. To perform the experiments in this kit, you get a special chassis, a power transformer, an iron-core choke (inductor), an ac line cord, resistors, capacitors, and miscellaneous hardware.

You explore the ac and dc characteristics of tubular and electrolytic capacitors, learning first-hand about "RC time constants," capacitor leakage, and capacitive voltage dividers. You see how a capacitor can pass ac current and block dc current.

You also examine in detail the characteristics of inductors in dc

and ac circuits and you determine how special resonant circuits can be made by combining inductors and capacitors. You also look at series and parallel resonance, seeing how these circuits can be tuned by changing component values.

By the time you've completed this kit, you'll have no trouble using your DMM to make both ac and dc current readings and resistor measurements.

The special design of the kits give you a complete breadboarding system for setting up and modifying prototype circuits, performing tests, and evaluating electronic components. You are introduced to the kind of high technology that's at the heart of today's astounding electronics revolution.

You'll perform an incredible array of tests and experiments. Even as you assemble the Lab, you perform 10 specific experiments demonstrating the nature of electronic principles; then you perform 10 experiments using parts that range from transistors through integrated circuits.

## **Specifications**

Power Supply:

±10V unregulated

±5V regulated, 50 mA max

Function Generator:

Switch selectable waveform:

Triangle: 10V p-p

Square: 0 to 5V TTL/CMOS compatible

Frequencies: 2Hz, 1Hz, 1kHz switch selectable.

Logic Indicators:

(4) identical LED circuits with drivers TTL/CMOS compatible inputs

Pushbuttons:

(1) debounced and buffered; TTL/CMOS compatible output

Slide Switch:

(1) 0 or +5V with 1K pullup resistor

Line Clock:

60 Hz buffered output, TTL/CMOS compatible

Other Features:

2-5k free potentiometer

SPDT meter switch:

Breadboarding socket, solderless connections for ICs and discrete components.

**Kit No. 4:** The successful operation of any computer depends on a correctly operating power supply. In this kit you study the fundamentals of power supplies.

You start by demonstrating counter-electromotive force and showing that it can be used to produce a high voltage. Then you study and compare the characteristics of the half-wave rectifier and the full-wave bridge rectifier circuit. You demonstrate the former's operation, see the effects of circuit defects, and compare its operation with that of a conventional full-wave rectifier. You go on to observe how a simple filter increases the dc component and decreases the ac component of a rectifier's output. You also take a good look at the operation of the pi filter and simulate a leaky electrolytic capacitor to observe the problem of the power factor in filters.

At this point you're ready to experiment with voltage doubler rectifiers and observe how they produce twice the dc output voltage of the basic half-wave and full-wave rectifiers. You go on to investigate the operation of shunt voltage regulators using reverse-biased Zener diodes. You examine the basic dc operation of npn and pnp transistors. You wrap up this kit by studying the series voltage regulator, and you actually assemble a dual polarity regulated power supply on an etched circuit board. You'll use this power supply in later parts of your course.

**Kit No. 5:** You're given detailed instruction for assembling the major portion of your kits. Their special design gives you a complete breadboarding system for setting up and modifying prototype circuits, performing tests, and evaluating components. Later, you'll use these kits to demonstrate the action of various computer functions.

Connecting components is quick and easy with the universal components matrix of your kits. You simply plug in component leads and you're ready to conduct experiments.

The experiments in this kit cover transistor fundamentals, and solid-state amplifiers and oscillators. You begin by demonstrating different methods of biasing npn and pnp transistors, and you build the basic amplifier types: common-emitter, common-collector, and common-base.

Now having experimented with the basic circuits, you are ready to use the transistor in more complex configurations. First you observe the operation of a phase splitter; then the operation of a common-emitter stage used as a switch and as a voltage amplifier. You learn about direct-coupled (DC) and resistance-capacitance

coupled (RC) amplifiers, and you construct and verify the operation of an LC oscillator circuit. You investigate the workings of two types of RC oscillators: phase-shift and multivibrator.

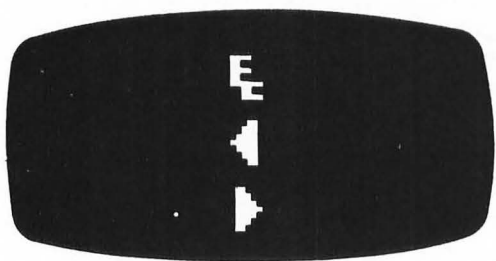
You'll quickly discover that the field-effect transistor plays an important role in computer electronics. Consequently, you round out this kit by experimenting with the methods of FET biasing and the three basic FET amplifiers: common-source, common-drift, and common-gate.

**Kit No. 6:** Now you're ready to move into high tech advanced electronic components and systems. To perform the experiments, you receive 2 dual operational amplifier (op amp) integrated circuits (ICs), 7 digital ICs, a 6-digit LED numeric display, an LED and phototransistor, a unijunction transistor (UJT), and a silicon controlled rectifier (SCR).

After you learn the basics of operational amplifier circuits you use a special dual op-amp IC to build a versatile function generator. This circuit provides you with a permanent signal source to furnish variable-frequency square and triangular waves. You use an LED level detector to see how these two waveforms differ.

You construct and examine digital logic circuits—circuits just like those that make up all computers. You work with the basic digital storage element, the flip-flop and see how it can be used as a frequency divider. You also learn how a binary-coded-decimal (BCD) counter works, and how to connect the BCD output to a decoder and numeric display unit.

# Appendix A



## Alphabetical Directory of Basic Reserved Words

The following material is courtesy of ATARI, Inc., a Warner Communications Company and is used with their permission.

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

RESERVED WORD:	ABBREVIATION:	BRIEF SUMMARY OF BASIC STATEMENT
ABS		Function returns absolute value (unsigned) of the variable or expression.
ADR		Function returns memory address of a string.
AND		Logical operator: Expression is true only if both subexpressions joined by and are true.
ASC		String function returns the numeric value of a single string character.
ATN		Function returns the arctangent of a number or expression in radians or degrees.
BYE	B.	Exit from BASIC and return to the resident operating system or console processor.
CLOAD	CLOA.	Loads data from Program Recorder into RAM.
CHR\$		String function returns a single string byte equivalent to a numeric value between 0 and 255 in ATASCII code.



RESERVED WORD:	ABBREVIATION:	BRIEF SUMMARY OF BASIC STATEMENT
CLOG		Function returns the base 10 logarithm of an expression.
CLOSE	CL.	I/O statement used to close a file at the conclusion of I/O operations.
CLR		The opposite of DIM: Undimensions all strings; matrices.
COLOR	C.	Chooses color register to be used in color graphics work.
COM		Same as DIM.
CONT	CON.	Continue. Causes a program to restart execution on the next line following use of the <b>BREAK</b> key or encountering a stop.
COS		Function returns the cosine of the variable or expression (degrees or radians).
CSAVE		Outputs data from RAM to the program recorder for tape storage.
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.	Statement deg tells computer to perform trigonometric functions in degrees instead of radians. (Default in radians.)
DIM	DI.	Reserves the specified amount of memory for matrix, array, or string. All string variables, arrays, matrices must be dimensioned with a dim statement.
DOS	DO.	Reserved word for disk operators. Causes the menu to be displayed. (See <i>DOS Manual</i> .)
DRAWTO	DR.	Draws a straight line between a plotted point and specified point.
END		Stops program execution; closes files; turns off sounds. Program may be restarted using cont. (Note: end may be used more than once in a program.)
ENTER	E.	I/O command used to store data or programs in untokenized (source) form.

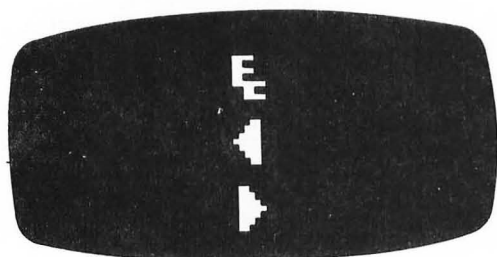
RESERVED WORD:	ABBREVIATION:	BRIEF SUMMARY OF BASIC STATEMENT
EXP		Function returns $e$ (2.7182818) raised to the specified power.
FOR	F.	Used with next to establish for/next loops. Introduces the range that the loop variable will operate in during the execution of loop.
FRE		Function returns the amount of remaining user memory (in bytes).
GET	GE.	Used mostly with disk operations to input a single byte of data.
GOSUB	GOS.	Branch to a subroutine beginning at the specified line number.
GOTO	G.	Unconditional branch to a specified line number.
GRAPHICS	GR.	Specifies which of the eight graphics modes is to be used. Gr.0 may be used to clear screen.
IF		Used to cause conditional branching or to execute another statement on the same line (only if the first expression is true).
INPUT	I.	Causes computer to ask for input from keyboard. Execution continues only when the <b>RETURN</b> key is pressed after inputting data.
INT		Function returns the next lowest whole integer below the specified value. Rounding is always downward, even when number is negative.
LEN		String function returns the length of the specified string in bytes or characters (1 byte contains 1 character).
LET	LE.	Assigns a value to a specific variable name. Let is optional in ATARI BASIC, and may be simply omitted.
LIST	L.	Display or otherwise output the program list.
LOAD	LO.	Input from disk, etc. into the computer.
LOCATE	LOC.	Graphics: Stores, in a specified variable, the value that controls a specified graphics point.

RESERVED WORD:	ABBREVIATION:	BRIEF SUMMARY OF BASIC STATEMENT
LOG		Function returns the natural logarithm of a number.
LPRINT	LP.	Command to line printer to print the specified message.
NEW		Erases all contents of user RAM.
NEXT	N.	Causes a for/next loop to terminate or continue depending on the particular variables or expressions. All loops are executed at least once.
NOT		A "1" is returned only if the expression is NOT true. If it is true, a "0" is returned.
NOTE	NO.	See <i>DOS/FMS Manual</i> . . . used only in disk operations.
ON		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.	Opens the specified file for input of output operations.
OR		Logical operator used between two expressions. If either one is true, a "1" is evaluated. A "0" results only if both are false.
PADDLE		Function returns position of the paddle game controller.
PEEK		Function returns decimal form of contents of specified memory location (RAM or ROM).
PLOT	PL.	Causes a single point to be plotted at the X,Y location specified.
POINT	P.	Used with disk operations only.
POKE	POK.	Insert the specified byte into the specified memory location. May be used only with RAM. Don't try to POKE ROM or you'll get an error.
POP		Removes the loop variable from the gosub stack. Used when departure from the loop is made in other than normal manner.
POSITION	POS.	Sets the cursor to the specified screen position.
PRINT	PR. or ?	I/O command causes output from the computer to the specified output device.

RESERVED WORD:	ABBREVIATION:	BRIEF SUMMARY OF BASIC STATEMENT
PTRIG		Function returns status of the trigger button on game controllers.
PUT	PU.	Causes output of a single byte of data from the computer to the specified device.
RAD		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 items in the data list and assign to specified variables.
REM	R. or. <b>SPACE</b>	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.	Return from subroutine to the statement immediately following the one in which gosub appeared.
RND		Function returns a random number between 0 and 1, but never 1.
RUN	RU.	Execute the program. Sets normal variables, to 0, undims array and string.
SAVE	S.	I/O statement causes data or program to be recorded on disk under filespec provided with save.
SETCOLOR	SE.	Store hue and luminance color data in a particular color register.
SGN		Function returns +1 if value is positive, 0 if zero, -1 if negative.
SIN		Function returns trigonometric sine of given value (deg or rad).
SOUND	SO.	Controls register, sound pitch, distortion, and volume of a tone or note.
SQR		Function returns the square root of the specified value.
STATUS	ST.	Calls status routine for specified device.

<b>RESERVED WORD:</b>	<b>ABBREVIATION:</b>	<b>BRIEF SUMMARY OF BASIC STATEMENT</b>
STEP		Used with for/next. Determines quality to be skipped between each pair of loop variable values.
STICK		Function returns position of stick game controller.
STRIG		Function returns 1 if stick trigger button not pressed, 0 if pressed.
STOP	STO.	Causes execution to stop, but does not close files or turn off sounds.
STR\$		Function returns a character string equal to numeric value given. For example: STR\$(65) returns 65 as a string.
THEN		Used with if: If expression is true, the then statements are executed. If the expression is false, control passes to next line.
TO		Used with for as in "FOR X = 1 TO 10". Separates the loop range expressions.
TRAP	T.	Takes control of program in case of an INPUT error and directs execution to a specified line number.
USR		Function returns results of a machine-language subroutine.
VAL		Function returns the equivalent numeric value of a string.
XIO	X.	General I/O statement used with disk operations (see <i>DOS/FMS Manual</i> ) and in graphics work (Fill).

# Appendix B



## Error Messages

ERROR CODE NO.	ERROR CODE MESSAGE
2	<b>Memory Insufficient</b> to store the statement or the new variable name or to dim a new string variable.
3	<b>Value Error:</b> A value expected to be a positive integer is negative, a value expected to be within a specified range is not.
4	<b>Too Many Variables:</b> A maximum of 128 different variable names is allowed. (See <b>Variable Name Limit.</b> )
5	<b>String Length Error:</b> Attempted to store beyond the DIMensioned string length.
6	<b>Out of Data Error:</b> Read statement requires more data items than supplied by DATA statement(s).
7	<b>Number greater than 32767:</b> Value is not a positive integer or is greater than 32767.
8	<b>Input Statement Error:</b> Attempted to input a non-numeric value into a numeric variable.
9	<b>Array or String DIM Error:</b> 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, or a reference has been made to an undimensioned array or string.
10	<b>Argument Stack Overflow:</b> There are too many gosubs or too large an expression.
11	<b>Floating Point Overflow/Underflow Error:</b> Attempted to divide by zero or refer to a number larger than $1 \times 10^{98}$ or smaller than $1 \times 10^{-99}$ .
12	<b>Line Not Found:</b> A gosub, goto, or then referenced a non-existent line number.
13	<b>No Matching For Statement:</b> A next was encountered without a previous for, or nested for/

## ERROR CODE NO.

## ERROR CODE MESSAGE

	next statement do not match properly. (Error is reported at the next statement, not at FOR).
14	<b>Line Too Long Error:</b> The statement is too complex or too long for BASIC to handle.
15	<b>Gosub or For Line Deleted:</b> A next or return statement was encountered and the corresponding for or gosub has been deleted since the last run.
16	<b>RETURN Error:</b> A return was encountered without a matching gosub.
17	<b>Garbage Error:</b> Execution of "garbage" (bad RAM bits) was attempted. This error 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	<b>Invalid String Character:</b> 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 drivers, printers, or other accessory devices. Further information is provided with the auxiliary hardware.
19	<b>Load program Too Long:</b> Insufficient memory remains to complete load.
20	<b>Device Number Larger</b> than 7 or Equal to 0.
21	<b>LOAD FILR ERROR:</b> Attempted to load a nonload file.
128	<b>BREAK Abort:</b> Use hit the <b>BREAK</b> key during I/O operation
129	<b>IOCB<sup>1</sup></b> already open
130	<b>Nonexistent Device</b> was specified.
131	<b>IOCB Write Only.</b> Read command to a write-only device (Printer).
132	<b>Invalid Command:</b> The command is invalid for this device.
133	<b>Device or File not Open:</b> No open specified for the device.
134	<b>Bad IOCB Number:</b> Illegal device number.
135	<b>IOCB Read Only Error:</b> Write command to a read-only device.
136	<b>EOF:</b> End of File read has been reached. (NOTE: This message may occur when using cassette files.)
137	<b>Truncated Record:</b> Attempt to read a record longer than 256 characters.
138	<b>Device Timeout.</b> Device doesn't respond.
139	<b>Device NAK:</b> Garbage at serial port or bad disk drive.
140	<b>Serial bus</b> input framing error.
141	<b>Cursor out of range</b> for particular mode.
142	<b>Serial bus data frame overrun.</b>
143	<b>Serial bus data frame checksum error.</b>
144	<b>Device done error</b> (invalid "done" byte): Attempt to write on a write-protected diskette.

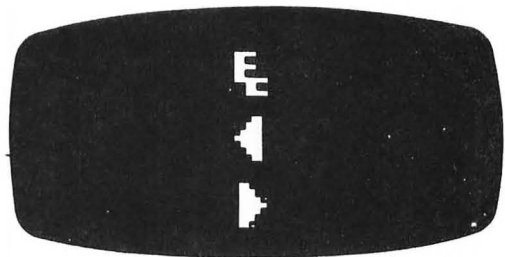
**ERROR  
CODE NO.****ERROR CODE MESSAGE**

145	<b>Read after write compare error</b> (disk handler) or bad screen mode handler.
146	<b>Function not implemented</b> in handler.
147	<b>Insufficient RAM</b> for operating selected graphics mode.
160	<b>Drive number error.</b>
161	<b>Too many OPEN files</b> (no sector buffer available).
162	<b>Disk full</b> (no free sectors)
163	<b>Unrecoverable system data I/O error.</b>
164	<b>File number mismatch:</b> Links on disk are messed up.
165	<b>File name error.</b>
166	<b>Point data length error.</b>
167	<b>File locked.</b>
168	<b>Command invalid</b> (special operation code).
169	<b>Directory full</b> (64 files).
170	<b>File not found.</b>
171	<b>Point invalid.</b>

<sup>1</sup>IOCB refers to input/output control block. The device number is the same as the IOCB number.




# Appendix C






## ATASCII Character Set

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
0	0		13	D		26	1A	
1	1		14	E		27	1B	
2	2		15	F		28	1C	
3	3		16	10		29	1D	
4	4		17	11		30	1E	
5	5		18	12		31	1F	
6	6		19	13		32	20	Space
7	7		20	14		33	21	!
8	8		21	15		34	22	"
9	9		22	16		35	23	#
10	A		23	17		36	24	\$
11	B		24	18		37	25	%
12	C		25	19		38	26	&

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
39	27	,	62	3E	>	85	55	u
40	28	(	63	3F	?	86	56	v
41	29	)	64	40	@	87	57	w
42	2A	*	65	41	A	88	58	x
43	2B	+	66	42	B	89	59	y
44	2C	,	67	43	C	90	5A	z
45	2D	-	68	44	D	91	5B	[
46	2E	.	69	45	E	92	5C	\
47	2F	/	70	46	F	93	5D	]
48	30	0	71	47	G	94	5E	^
49	31	1	72	48	H	95	5F	_
50	32	2	73	49	I	96	60	
51	33	3	74	4A	J	97	61	a
52	34	4	75	4B	K	98	62	b
53	35	5	76	4C	L	99	63	c
54	36	6	77	4D	M	100	64	d
55	37	7	78	4E	N	101	65	e
56	38	8	79	4F	O	102	66	f
57	39	9	80	50	P	103	67	g
58	3A	:	81	51	Q	104	68	h
59	3B	;	82	52	R	105	69	i
60	3C	<	83	53	S	106	6A	j
61	3D	=	84	54	T	107	6B	k

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
108	6C	l	131	83		154	9A	
109	6D	m	132	84		155	9B	(EOL) RETURN
110	6E	n	133	85		156	9C	↑
111	6F	o	134	86		157	9D	↓
112	70	p	135	87		158	9E	←
113	71	q	136	88		159	9F	→
114	72	r	137	89		160	A0	
115	73	s	138	8A		161	A1	
116	74	t	139	8B		162	A2	
117	75	u	140	8C		163	A3	
118	76	v	141	8D		164	A4	
119	77	w	142	8E		165	A5	
120	78	x	143	8F		166	A6	
121	79	y	144	90		167	A7	
122	7A	z	145	91		168	A8	
123	7B	♣	146	92		169	A9	
124	7C		147	93		170	AA	
125	7D	↖	148	94		171	AB	
126	7E	↗	149	95		172	AC	
127	7F	↘	150	96		173	AD	
128	80		151	97		174	AE	
129	81		152	98		175	AF	
130	82		153	99		176	B0	

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
177	B1		200	C8		223	DF	
178	B2		201	C9		224	E0	
179	B3		202	CA		225	E1	
180	B4		203	CB		226	E2	
181	B5		204	CC		227	E3	
182	B6		205	CD		228	E4	
183	B7		206	CE		229	E5	
184	B8		207	CF		230	E6	
185	B9		208	D0		231	E7	
186	BA		209	D1		232	E8	
187	BB		210	D2		233	E9	
188	BC		211	D3		234	EA	
189	BD		212	D4		235	EB	
190	BE		213	D5		236	EC	
191	BF		214	D6		237	ED	
192	C0		215	D7		238	EE	
193	C1		216	D8		239	EF	
194	C2		217	D9		240	F0	
195	C3		218	DA		241	F1	
196	C4		219	DB		242	F2	
197	C5		220	DC		243	F3	
198	C6		221	DD		244	F4	
199	C7		222	DE		245	F5	

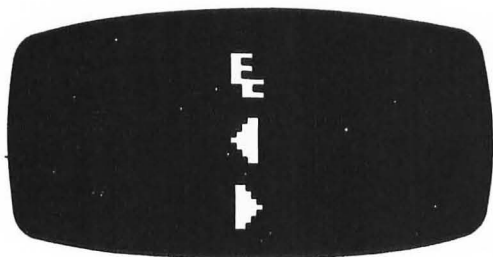
DECIMAL CODE	HEX ADECIMAL CODE	CHARACTER	DECIMAL CODE	HEX ADECIMAL CODE	CHARACTER	DECIMAL CODE	HEX ADECIMAL CODE	CHARACTER
246	F6		250	FA		254	FE	 (Delete character)
247	F7		251	FB		255	FF	 (Insert character)
248	F8		252	FC				
249	F9		253	FD	 (Buzzer)			

See Appendix H for a user program that performs decimal hexadecimal conversion.

**Notes:**

1. ATASCII stands for "ATARI ASCII". Letters and numbers have the same values as those in ASCII, but some of the special characters are different.
2. Except as shown, characters from 128-255 are reverse colors of 1 to 127.
3. Add 32 to upper case code to get lower case code for same letter.
4. To get ATASCII code, tell computer (direct mode) to PRINT ASC ("\_\_\_\_\_") Fill blank with letter, character, or number of code. Must use the quotes!









# Appendix D



## ATARI 400/800 Memory Map

ADDRESS		CONTENTS
Decimal	Hexadecimal	
65535 57344	FFFF E000	OPERATING SYSTEM ROM
57343 55296	DFFF D800	FLOATING POINT ROM
55295 53248	D7FF D000	HARDWARE REGISTERS
53247 49152	CFFF C000	NOT USED
49151 40960	BFFF A000	CARTRIDGE SLOT A (may be RAM if no A or B cartridge)
40959 32768	9FFF 8000	CARTRIDGE SLOT B (may be RAM if no B cartridge)
32767	7FFF	(7FFF if 32K system) DISPLAY DATA (size varies)

← RAMTOP (MSB)

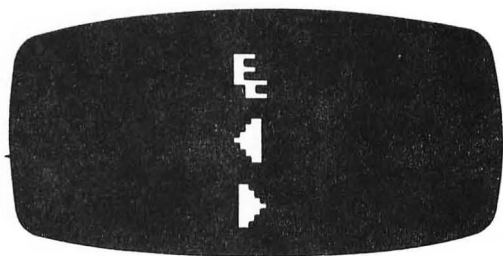
ADDRESS		CONTENTS
Decimal	Hexadecimal	
31755	7C1F	DISPLAY LIST (size varies) (7C1F if 32K system, (GRAPHICS 0) 
		 OS MEMTOP
		FREE RAM (size varies) 
		 BASIC MEMTOP
		BASIC program, buffers, tables, run-time stack. (2A80 if DOS, may vary)
10880	2A80	 
		 OS MEMLO
		 BASIC LOMEM
10879	2A7F	DISK OPERATING SYSTEM (2A7F-700)
9856	2680	DISK I/O BUFFERS (current DOS)
9855	267F	DISK OPERATING SYSTEM RAM (current DOS)
4864	1300	
4863	12FF	FILE MANAGEMENT SYSTEM RAM (current DOS)
1792	700	
1791	6FF	FREE RAM
1536	600	
1535	5FF	FLOATING POINT (used by BASIC)
1406	57E	
1405	57D	BASIC CARTRIDGE
1152	480	
1151	47F	] OPERATING SYSTEM RAM (47F-200)
1021	3FD	
		CASSETTE BUFFER
1020	3FC	RESERVED
1000	3E8	
999	3E7	PRINTER BUFFER
960	3C0	
959	3BF	] IOCB's
832	340	
831	33F	] MISCELLANEOUS OS VARIABLES
512	200	
511	1FF	HARDWARE STACK
256	100	
255	FF	PAGE ZERO
212	D4	FLOATING POINT (used by BASIC)

ADDRESS		CONTENTS
Decimal	Hexadecimal	
211	D3	BASIC or CARTRIDGE PROGRAM
210	D2	
209	D1	FREE BASIC RAM
208	D0	
207	CF	FREE BASIC AND ASSEMBLER RAM
203	CB	
202	CA	<div> <div> FREE ASSEMBLER RAM  ASSEMBLER ZERO PAGE </div> <div> } BASIC  ZERO PAGE </div> </div>
176	B0	
128	80	
127	7F	OPERATING SYSTEM RAM
0	0	

As the addresses for the top of RAM, OS, and BASIC and the ends of OS and BASIC vary according to the amount of memory, these addresses are indicated by pointers.



# Appendix E



## Derived Functions

### Derived Functions

Secant

Cosecant

Inverse Sine

Inverse Cosine

Inverse Secant

Inverse Cosecant

Inverse Cotangent

Hyperbolic Sine

Hyperbolic Cosine

Hyperbolic Tangent

Hyperbolic Secant

Hyperbolic Cosecant

Hyperbolic Cotangent

Inverse Hyperbolic Sine

Inverse Hyperbolic Cosine

Inverse Hyperbolic Tangent

Inverse Hyperbolic Secant

Inverse Hyperbolic Cosecant

Inverse Hyperbolic Cotangent

### Derived Functions in Terms of ATARI Functions

$\text{SEC}(X) = 1/\text{COS}(X)$

$\text{CSC}(X) = 1/\text{SIN}(X)$

$\text{ARCSIN}(X) = \text{ATN}(X/\text{SQR}(-X*X+1))$

$\text{ARCCOS}(X) = -\text{ATN}(X/\text{SQR}(-X*X+1)) + \text{CONSTANT}$

$\text{ARSEC}(X) = \text{ATN}(\text{SQR}(X*X-1)) + (\text{SGN}(X-1)) * \text{CONSTANT}$

$\text{ARCCSC}(X) = \text{ATN}(1/\text{SQR}(X*X-1)) + (\text{SGN}(X-1)) * \text{CONSTANT}$

$\text{ARCCOT}(X) = \text{ATN}(X) + \text{CONSTANT}$

$\text{SINH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/2$

$\text{COSH}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$

$\text{TANH}(X) = -\text{EXP}(-X)/(\text{EXP}(X) + \text{EXP}(-X)) * 2 + 1$

$\text{SECH}(X) = 2/(\text{EXP}(X) + \text{EXP}(-X))$

$\text{CSCH}(X) = 2/(\text{EXP}(X) - \text{EXP}(-X))$

$\text{COTH}(X) = \text{EXP}(-X)/(\text{EXP}(X) - \text{EXP}(-X)) * 2 + 1$

$\text{ARCSINH}(X) = \text{LOG}(X + \text{SQR}(X*X+1))$

$\text{ARCCOSH}(X) = \text{LOG}(X + \text{SQR}(X*X-1))$

$\text{ARCTANH}(X) = \text{LOG}((1+X)/(1-X))/2$

$\text{ARCSECH}(X) = \text{LOG}((\text{SQR}(-X*X+1)+1)/X)$

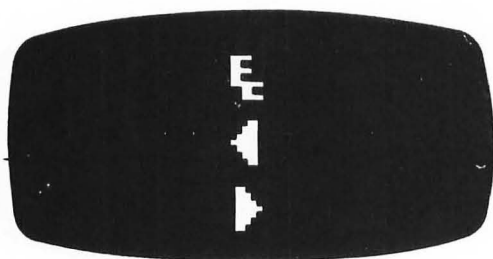
$\text{ARCCSCH}(X) = \text{LOG}((\text{SGN}(X) * \text{SQR}(X*X+1)+1)/X)$

$\text{ARCCOTH}(X) = \text{LOG}((X+1)/(X-1))/2$

### Notes:

1. If in RAD (default) mode, constant = 1.57079633  
If in DEG mode, constant = 90.
2. In this chart, the variable X in parentheses represents the value or expression to be evaluated by the derived function. Obviously, any variable name is permissible, as long as it represents the number or expression to be evaluated.

# Appendix F

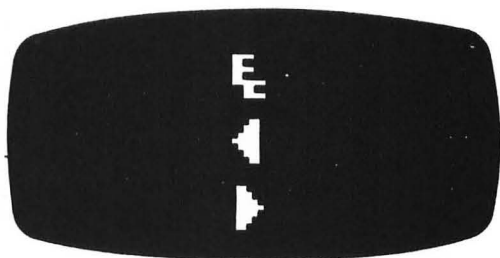


## Printed Versions of Control Characters

The cursor and screen control characters can be placed in a string in a program or used as a direct mode statement by pressing the **ESC** key before entering the character from the keyboard. This causes the special symbols which are shown below to be displayed.

PRESS		PRESS	SEE THIS
ESC	ESC		
ESC	DELETE BACKS		
ESC	ESC TAB		
PRESS	PRESS AND HOLD	PRESS	
ESC	ESC	ESC	
ESC	ESC	ESC	
ESC	ESC	ESC	
ESC	ESC	ESC	
OR			
ESC	ESC	ESC	
ESC	ESC	ESC	
ESC	ESC	ESC	
ESC	ESC	ESC	
ESC	ESC	ESC	
ESC	ESC	ESC	
ESC	ESC	ESC	
ESC	ESC	ESC	
ESC	ESC	ESC	

## Appendix G



### Memory Locations

*Note:* Many of these locations are of primary interest to expert programmers and are included here as a convenience. The labels given are used by ATARI programmers to make programs more readable.

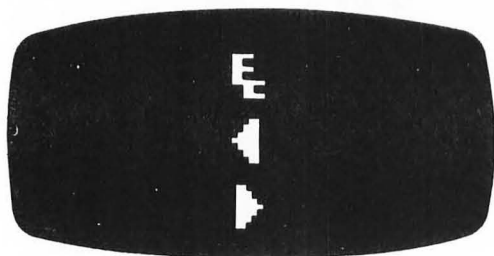
LABEL	DECIMAL LOCATION	HEXADECIMAL LOCATION	COMMENTS AND DESCRIPTIONS
APPMHI	14,15	DE	Highest location used by BASIC (LSB, MSB)
RTCLOK	18,19,20	12,13,14	TV frame counter (1/60 sec.) (LSB, NSB, MSB)
SOUNDR	65 77	41	Noisy I/O Flag (0=quiet) Attract Mode Flag (128=Attract mode)
LMARGIN, RMARGIN	82,83	52,53	Left, Right Margin (Defaults 2, 39)
ROWCRS	84	54	Current cursor row (graphics window).
COLCRS	85,86	55,56	Current cursor column (graphics window)
OLDROW	90	5A	Previous cursor row (graphics window).
OLDCOL	91,92	5B	Previous cursor column (graphics window).
	93	5C	Data under cursor (graphics window unless mode 0).
NEWROW	96	60	Cursor row to which drawto will go.
NEWCOL	97,98	61,62	Cursor column to which drawto goes.
RAMTOP	106	6A	Actual top of memory (number of pages).

LABEL	DECIMAL LOCATION	HEXADECIMAL LOCATION	COMMENTS AND DESCRIPTIONS
LOMEM	128,129	80,81	BASIC low memory pointer.
MEMTOP	144,145	90,91	BASIC top of memory pointer.
STOPLN	186,187	BA,BB	Line number at which stop or trap occurred (2-byte binary number).
ERRSAV	195	C3	Error number.
PTABW	201	C9	Print tab width (defaults to 10)
FR0	212,213	D4,D5	Low and high bytes of value to be returned to BASIC from USR function.
RADFLG	251	FB	RAD/DEG flag (0 = radians, 6 = degrees).
LPENH	564	234	Light Pen* Horizontal value.
LPENV	565	235	Light Pen* Vertical value.
TXTROW	656	290	Cursor row (text window)
TXTCOL	657,658	291,292	Cursor column (text window)
COLOR0	708	2C4	Color Register 0
COLOR1	709	2C5	Color Register 1
COLOR2	710	2C6	Color Register 2
COLOR3	711	2C7	Color Register 3
COLOR4	712	2C8	Color Register 4
MEMTOP	741,742	2E5,2E6	OS top of available user memory pointer (LSB, MSB)
MEMLO	743,744	2E7,2E8	OS low memory pointer
CRSINH	752	2F0	Cursor inhibit (0 = cursor, on, 1 = cursor off)
CHACT	755	2F3	Character mode register 4 = vertical reflect; 2 = normal; 1 = blank)
CHBAS	756	2F4	Character base register (defaults to 224) (224 = upper case, 226 = lower case characters)
ATACHR	763	2FB	Last ATASCII character.
CH	764	2FC	Last keyboard key pressed; internal code; (255 clears character).
FILDAT	765	2FD	Fill data for graphics Fill (XIO).
DSPFLG	766	2FE	Display Flag (1 = display control character).
SSFLAG	767	2FF	Start/Stop flag for paging (0 = normal listing) Set by <b>CTRL</b> 1.
HATABS	794	31A	Handler address table (3 bytes/handler)

LABEL	DECIMAL LOCATION	HEXADECIMAL LOCATION	COMMENTS AND DESCRIPTION
IOCB	832	340	I/O control blocks (16 bytes/IOCB)
CONSOL	1664-1791 53279	680-6FE D01F	Spare RAM Console switches (bit 2 = Option; bit 1 = Select; bit 0 = Start. Poke 53279, 0 before reading. 0 = switch pressed.)
PORTA	54016	D300	PIA Port A Controller Jack I/O ports.
PORTB	54017	D301	PIA Port B Initialized to hex 3C.
PACTL	54018	D302	Port A Control Register (on Program Recorder 52 = ON, 60 = OFF).
PBCTL	54019	D303	Port B control register.
SKCTL	53775	D20F	Serial Port control register. Bit 2 = 0 (last key still pressed).

\* Future product.

## Appendix H



## Glossary of Microcomputer Terms

**Address:** A location in memory, usually specified by a two-byte number in hexadecimal or decimal format.

**Algorithm:** A precisely defined set of rules or a structured procedure which provides the solution to a problem in a finite number of steps. An example follows:

Compute the value of a polynomial of the form

$$f(9x) = ax^n + bx^{n-1} + cx^{n-2} \dots$$

For  $n=5$  we can write the following algorithm:

$$f(x) = (((((a)x+b)x+c)x+d)x+e)x+f$$

This algorithm indicates that the computation can be done by repeating a multiply-and-add step five times.

**Alphanumeric:** The capital letters A-Z and the numbers 0-9, and/or combinations of letters and numbers. Specifically excludes graphics symbols, punctuation marks, and other special characters.

**ALU:** Abbreviation for arithmetic-and-logic unit.

**Analog computer:** A computer that uses variables represented by voltages, currents, or other parameters whose values are analogous to the quantitative magnitude of the variables.

**Analog-to-digital conversion:** The process of representing precisely a varying voltage or current by a series of discrete pulses when the varying voltage or current itself is an analog of some other form of information. Analog-to-digital conversion techniques are required to convert any information that is in

analog form into the digital form required by the microcomputer. Analog data is frequently encountered in data acquisition as voltage from a transducer, potentiometer, or other sensor of physical data.

**Analyzer:** A routine to analyze a program. It usually consists of summarizing references to storage locations and tracing jump sequences.

**AND:** A logic operation that states that if A and B are both statements, then A AND B is true if both statements are true, but it is false if either one or both statements are false. Truth is usually expressed by the value 1, while 0 is used to indicate a false state. The AND operator is usually represented by a centered dot

$$A \cdot B$$

or by no sign

$$AB$$

An inverted u is sometimes used to denote the logical product

$$A \text{ n } B$$

Finally, the standard multiplication sign may be used to express the AND function

$$A \times B$$

**Array:** A one dimensional set of elements that can be referenced one at a time or as a complete list by using the array variable name and one subscript. Thus the array B, element number 10 would be referred to as B(10). Note that string arrays are not supported by BASIC, and that all arrays must be DIMensioned. A matrix is a two dimensional array.

**ASCII keyboard:** A typewriter terminal keyboard laid out in a specific format and containing the character symbol buttons and function switches required to generate signals representing the 127 different character variations of the American Standard Code for Information Interchange.

**Assembler:** A computer program that is used to translate symbolic language into machine language. A typical program supplied by a microcomputer manufacturer is loaded using PROM or ROM chips and then executed to perform the assembly operation.

**Assembly language:** A language that uses words, statements, and phrases to produce machine instructions. Most micro-processor programs are written as a series of source statements that are then translated using an assembly language into machine code.

## ATASCII—Binary Load

**ATASCII:** Stands for ATARI American Standard Code for Information Interchange. It is the method of coding used to store text data. In ATASCII each character and graphics symbol, as well as most of the control keys, has a number assigned to represent it. The number is a one-byte code (decimal 0-255).

**Backup DOS Disk:** An exact copy of original master DOS diskette. Always keep backups of your DOS master diskette and of all important data diskettes.

**BASIC:** High-level programming language. Acronym for Beginner's All-purpose Symbolic Instruction Code. BASIC is always written using all capital letters. Developed by Mssrs. Kemeny and Kurtz at Dartmouth College in 1963.

**Baud Rate:** Signalling speed or speed of information interchange in bits per second.

**BCD:** Abbreviation for binary-coded decimal, a method of coding in which each decimal digit is coded into a separate 4-bit word. BCD is also known as the 8421 code due to the weight assigned to each 4-bit word. The numbers used to count to ten are as follows:

DECIMAL	BINARY
0	0
1	1
BINARY	DECIMAL
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10

**Binary:** A number system using the base two. Thus the only possible digits are 0 and 1, which may be used in a computer to represent true and false, on and off, etc.

**Binary Arithmetic:** Mathematical operations performed using the binary digits of 0 and 1.

**Binary Load:** Loading a binary machine-language object file into the computer memory.



**Binary Save:** Saving a binary machine-language object file onto a disk drive or program recorder.

**Bit:** Short for Binary Digit. A bit can be thought of as representing 0 or 1, true or false, whether a circuit is on or off, or any other type of two-possibility concept. A bit is the smallest unit of data with which a computer can work. The word *bit* is a blend word formed from “binary digit,” a unit of information equal to one binary decision. It can be a single character in a binary number, a single pulse in a coded group of pulses, or a unit of information capacity.

When used as a unit of information capacity, the capacity in bits is equal to the logarithm to the base two of the number of possible states available. In a memory, for example, each element is capable of representing a zero or a one at any instant, and the total number of ones and zeros at any instant is the capacity of the memory.

**Boot:** This is the initialization program that “sets up” the computer when it is powered up. At conclusion of the boot (or after “booting up”), the computer is capable of loading and executing higher-level programs.

**Branch:** 1. ATARI BASIC executes a program in order of line numbers. This execution sequence can be altered by the programmer, and the program can be told to skip over a certain number of lines or return to a line earlier in the program. This contrived change in execution sequence is called “branching”. 2. The selection of one or more paths in the flow of data or signals through a system or stage. Selection of a path is based upon some criterion to allow a decision to be reached. The instructions used to mechanize the selection are sometimes called branch instructions. Transfer of control and jump operations are also used in this context. In a microprocessor program, branching is done on the basis of computed results causing modification of the function or program sequence. The usual modification can be classed as (1) a change of the program direction, or (2) a departure from the normal sequence.

**Break:** To interrupt execution of a program. Pressing the break key causes a break in execution.

**Buffer:** A temporary storage area in RAM used to hold data for further processing, input/output operations, etc.

**Bug:** A usually elusive error in a program, circuit, or machine.

## **Byte—Chip**

**Byte:** Usually eight bits (enough to represent the decimal number 255 or 11111111 in binary notation). A byte of data can be used to represent an ATASCII character or a number in the range of 0 to 255. Byte is a generic term developed by IBM to indicate a measureable number of consecutive binary digits which are usually operated upon as a unit. Bytes of eight bits representing either one character or two numerals are most often encountered. Words in some systems are divided into high and low bytes and the byte addresses are either odd or even numbers. The high bytes are stored at the odd-numbered locations and the low bytes at the even-numbered ones.

**Cassette:** A magnetic-tape housing that contains a tape of a specific length. The cassette includes the supply reel, takeup reel, head pressure pad, and a slot for the capstan drive.

**Central Processing Unit (CPU):** In microcomputers such as the ATARI systems, these are also called microprocessors or MPU. At one time, the CPU was that portion of any computer that controlled the memory and peripherals. Now the CPU or MPU is usually found on a single integrated circuit or “chip” (in ATARI’s case a 6502 microprocessor chip).

**Character:** A letter, numeral, or other symbol that is used to express information. It is usually part of an ordered set and may be graphic. Letters from English and other languages, any form of numeral, all punctuation marks, and other formatting symbols are all considered characters.

**Character Code:** An ordered pattern of bits that are assigned in a particular system to represent characters. Baudot and ASCII are character codes.

**Chip:** In its basic form, a chip is the integrated circuit inside an IC housing. Those used in microcomputers typically utilize large-scale integration techniques and up to 10,000 transistors have been placed on chips of 6 mm square! The chips are usually mounted in dual-inline packages which may have up to 40 pins for mounting on circuit boards.

A microcomputer chip usually includes the arithmetic logic unit (ALU), general purpose register, and bus controls. The chip of chips may also be combined with memory and input/output chips to form a computer system that will fit on a single circuit board.

**CIO:** Central Input/Output Subsystem. The part of the operating system that handles input/output.

**CLOSE:** To terminate access to a disk file. Before further access to the file, it must be opened again. See **Open**.

**Code:** 1. (noun) Instructions written in a language understood by a computer.

**Code:** 2. (verb) To write or form a routine in a computer program using a system of rules and ordered characters which represent symbols and characters from a different character set.

**Coding:** The process of converting program flowcharts into the desired computer language.

**Command:** An instruction to the computer that is executed immediately. A good example is the BASIC command **run**. (See **Statement**.)

**Compiler:** A coding or programming system that inputs source language data and outputs a program either in assembly or machine language. The compiler provides a language that requires fewer statements for algorithm wiring and eliminates the requirement for detailed coding to control loops, access data structures, and write complex formulas and functions.

**Computer:** Any device that can receive and then follow instructions to manipulate information. Both the instructions and the information may be varied from moment to moment. The distinction between a computer and a programmable calculator lies in the computer's ability to manipulate text as well as numbers. Most calculators can only handle numbers.

**Computer Code:** The machine language or code used with a computer system.

**Computer Program:** The series of instructions and statements prepared in order to solve a specific problem or achieve a certain result.

**Concatenation:** The process of joining two or more strings together to form one longer string.

**Console:** The part of a computer system that is used for communication between the computer operator and the CPU. The computer console may contain a start key, a stop key, a power control switch, sense switches, and register lights. Other console functions allow manual control, error correction, and status determination for counter and storage circuits. Some consoles allow the programmer to debug the program by slowly stepping the

## **Console—CPU**

machine through each instruction and observing the status indicators; other consoles include a CRT terminal which may have pictorial capabilities.

**Constant:** A fixed value or an item of data that does not vary. Constants are those quantities or messages which are available as data for the program and are not subject to change with time. A constant can be one character or a group of characters which represent a value and are used to identify, measure, or compare.

**Control Characters:** Characters produced by holding down the key labeled CTRL while simultaneously pressing another key.

**Control Console:** A console that enables the operator to control and monitor all processing functions from a central location. The control console usually has a typewriter or other keyboard to allow communication with the processor.

**Controller:** A unit which operates automatically to regulate the performance of a controlled variable or system. Controllers use the data from the input and the output of a device to obtain the maximum and most efficient control of the device or process. Controllers are also used in computer systems for generation of signals to interface the CPU with memory and peripheral equipment. A typical controller chip performs all the control functions required for the flow of data at the interface.

**Control Program:** A sequence of instructions used to guide the CPU through operations. A control program in a microprocessor is usually stored in ROM where it can be accessed, but not erased by the CPU.

**Core Memory:** A memory that uses toroids in matrix arrays for storing binary digits. A core memory is a programmable random-access memory that uses ferromagnetic storage techniques.

**Coupler:** A device which transfers signals and has its input and output isolated electrically. Also known as an isolator, a coupler usually consists of a light-emitting diode (LED) and a light sensor. The input signal activates the LED, which in turn forces current through the output light sensor. When the two devices are separated by an air gap rather than glass or plastic, the coupler can be used to sense motion and encode cards, plates, and slotted disks.

**CPU:** Abbreviation for central processing unit. A primary unit of the computer system which controls the interpretation and

execution of instruction. A CPU may consist of registers, computational circuits such as the arithmetic and logic unit, control circuits, and input/output ports. The register may include accumulators, index registers, and perhaps stack registers. All registers are treated as internal memory.

**CRT:** Abbreviation for “cathode-ray tube” (the tube used in a TV set). In practice, this is often used to describe the television receiver used to display computer output. Also called a “monitor”. CRTs are used for both display and storage in computer systems. As a display, the CRT can be operated in the point mode to allow points to be established and displayed on the screen. CRT storage uses the electron beam to sense the presence or absence of spots on the CRT screen.

**Cursor:** A square displayed on the TV monitor that shows where the next typed character will be displayed. It is a manually controllable pointer and can be employed to indicate a character to be changed or corrected, or a position where data is to be entered via the keyboard. The cursor may also take the form of an underscore, an undertext caret, or an arrow.

**Daisy Wheel:** A plastic or metal print wheel on certain types of impact printers, so called because of its symmetrical “petals.”

**Data:** Information of any kind.

**Debug:** The process of locating and correcting mistakes and errors in a program. Debugging involves the running and checking of programs to detect any errors. One typical method of debugging includes running a similar problem with a known answer through the system checkout. Debugging aids are available to allow quick development of microcomputer programs.

**Decimal:** A number base system using the digits 0 through 9. Decimal numbers are stored in binary-coded-decimal format in the computer. See Bit, Hexadecimal, and Octal.

**Decision:** A determination of future action. A decision in computer systems usually involves a comparison to determine the existence or nonexistence of a specific condition before taking a specific succeeding action. The action may involve a conditional jump or transfer of control. The comparison may take place between words or numerical characters in registers or other temporary storage.

**Decode:** The act of applying a set of data rules to restore a previous representation, or to reverse a previous encoding operation.

## Decoder—Disk file

**Decoder:** A device which determines the meaning of a set of data and usually initiates some action based on the meaning. A decoder may be a matrix of switching elements used to select one or more channels according to the combination of input signals. Many decoder chips can be expanded so that each decoder drives eight or more other decoders for large system applications.

**Default:** A mode or condition “assumed” by the computer until it is told to do something else. For example, The ATARI will “default” to screen and keyboard unless told to use other I/O devices. It will also default to GRAPHICS 0 until another graphics mode is entered.

**Delimiter:** A character that marks the start or finish of a data item but is not part of the data. For example, the quotation marks (") are used by most BASIC systems to delimit strings.

**Destination:** The device or address that receives data during an exchange of information (and especially an I/O exchange). See Source.

**Digital:** Information that can be represented by a collection of bits. Virtually all modern computers, especially microcomputers, use the digital approach.

**Digital-to-analog converter:** A device for converting digital signals into continuous analog signals. Digital-to-analog converters usually buffer the input so that the output remains the same until the input changes. Units are available with up to 16 channels that can operate at 10,000 samples per second with a 100-micro-second conversion time.

**Directory:** A listing of the files contained on a diskette.

**Discrete programming:** A class of procedures used in operations research for locating the maximum and minimum values of a function. All variables must have integer values or similar constraints.

**Diskette:** A small disk. A record/playback medium like tape, but made in the shape of a flat disk that is placed inside a stiff envelope for protection. The advantage of the disk over cassette or other tape for memory storage is that access to any part of the disk is virtually immediate. The ATARI 800 Personal Computer System can control up to 4 diskette drive peripherals simultaneously. In this book, disk and diskette are used interchangeably.

**Disk file:** An associated set of records of a similar format which can be grounded by a common label and may be accessed as a unit.

**Disk pack:** Portable direct-address storage units which use magnetic disks. Disk packs are mounted on disk storage drives for read and write operations.

**Disk storage:** A memory system that stores data on magnetic disks. The disks require a disk drive. Floppy disks, which are used on the ATARI, are flexible and can store up to 300,000 words.

**Display:** A visual representation of data, which may take the form of numerals, alphanumeric characters, or graphics. A display unit can provide viewing access into the operation of the microcomputer system.

**DOS:** Abbreviation for “disk operating system”. The software or programs which facilitate use of a disk-drive system.

**DOS.SYS:** Filename reserved by Disk Operating System.

**Drive Number:** An integer from 1 to 4 that specifies the drive to be used.

**Drive Specification or Drivespec:** Part of the filespec that tells the computer which disk drive to access. If this is omitted the computer will assume drive 1.

**Edit:** To make corrections or changes in a program or data.

**Electronic digital computer:** A machine which uses electronic circuitry to perform arithmetic and logic operations by means of an internally or externally stored program of machine instructions.

**Electronic pen:** A stylus that is used with a CRT display for input to the system computer. Also called a light pen, the stylus signals the computer with electronic pulses which the computer interprets.

**Encoder:** Any device or circuit that provides an enabling code to permit an otherwise unusable system or device to be used in an environment where the code is required. In a selective calling communications system, an encoder may be a simple tone oscillator of a specific frequency.

**End of File:** A marker that tells the computer that the end of a certain file on disk has been reached.

**Entry Point:** The address where execution of a machine-language program or routine is to begin. Also called the transfer address.

**Erase:** To clear or obliterate information in a storage medium. Erasing results in the replacement of all digits with zeros in

## **Erase—Flowchart**

magnetic storage. Erasing in some EPROMs is done with a specified level of ultraviolet light.

**Execute:** To do what a command or program specifies. To run a program or portion thereof.

**Expression:** A combination of variables, numbers, and operators (+, −, etc.) that can be evaluated to a single quantity. The quantity may be a string (sexp) or a number (aexp).

**Fetch:** 1. To locate and load a program from storage, as in bringing a program phase into main storage for execution from the memory library. 2. That portion of a computer cycle from which the location of the next instruction is obtained. A fetch can also be used to retrieve phases of a program and load them into main storage, or transfer control to a system loader.

**File:** An organized collection of related data. A file is the largest grouping of information that can be addressed with a single name. For example, a BASIC program is stored on diskette as a particular file, and may be addressed by the statements save or load (among others).

**Filename:** The alphanumeric characters used to identify a file. A total of 8 numbers and/or letters may be used. The first of these must be a letter.

**Filename Extender or Extension:** From 1 to 3 additional characters used following a period (required if the extender is used) after the filename. For example, in the filename PHON-LIST.BAS, the letters "BAS" comprise the extender.

**Filespec:** Abbreviation for file specification. A sequence of characters which specifies a particular device and filename. Do not create two files with exactly the same filespec. If you do, and they are both stored on the same diskette, you will not be able to access the second file, and the DOS functions of delete file, rename file, and copy file will act on both files.

**Flip-flop:** A circuit that is capable of assuming either one of two stable states; a bistable multivibrator. It will assume a given state depending upon the previous history of the inputs.

**Floppy Disks:** A magnetic storage medium that uses flexible disks that resemble phonograph records. Each one can provide random access storage for as much as 163,000 or more bytes, (double density) and are used to replace cassettes in many applications.

**Flowchart:** A graphical representation of the definition or solution



of a problem, in which symbols are used to represent functions, operations, and flow. It can contain all of the logical steps in a routine or program in order to allow the designer to conceptualize and visualize each step. It defines the major phases of the processing, as well as the path to problem solution.

**Format:** 1. (noun) A predetermined arrangement of data, words, letters, characters, files, etc. 2. (verb) To specify the form in which something is to appear. 3. To organize a new or magnetically (bulk) erased diskette into tracks and sectors. When formatted, each diskette contains 40 circular tracks, with 18 sectors per track. Each sector can store up to 128 bytes of data (single density), or 256 bytes of data (double density).

**Gap:** A space or interval which appears between items of data. A magnetic gap refers to the air space in the magnetic circuit. A head gap refers to the separation between the pole pieces of a magnetic recording head. A data gap may appear as an interval of space or time to indicate the end of a word, record, or file on tape, or it may be the complete absence of information for a length of time or space on the recording medium.

**Gate:** A circuit or device having one output and one or more inputs with the output state completely determined by the previous and present states of the inputs. A gate can also be a trigger used to allow the passage of other signals through a circuit. Logical gates can take many forms.

**Get:** To locate, fetch, and transfer an item from storage, (the activity required to develop or make a record from an input file available), or to obtain or extract a coded value from a field. The get command might be used to obtain a numerical value from a series of decimal digits.

**Goto:** A multilanguage statement that directs the computer to leave the current sequence of instructions and begin operating at another point in the program. A typical BASIC goto instruction might be:

goto 5; (the next statement to be executed is number 5)

**Graphic:** 1. Any assembly of symbols or characters that is used to denote any concept, configuration, or idea non-textually. 2. Any symbol produced by printing, drawing, handwriting, etc.

**Graphic display:** A nontextual display which reproduces data on a video screen, panel, or page.

## **Halt—Information bits**

**Halt:** A condition occurring after the sequence of operations in a program stops. A halt may be due to a halt (stop) instruction, an unexpected halt, or an interrupt. The program would normally continue after the halt unless a “drop-dead” halt occurs. In this case there is no recovery. The drop-dead halt may be a part of the program designed to shut down the system, or it may be due to an error in programming such as division by zero, or transfer to a nonexistent instruction.

**Hard Copy:** Printed output as opposed to temporary TV monitor display.

**Hardware:** The physical components of a computer system, including all electronic and electromechanical devices and connections.

**Hardware assembler:** An assembler that usually consists of PROMs which are mounted on simulation boards. A hardware assembler allows the prototype unit to assemble its own programs.

**Hexadecimal or Hex:** Number base system using 16 alphanumeric characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

**Hold:** A condition which suspends operation of a system or circuit for a specified time.

**Hold instruction:** An instruction which causes data called from storage to be retained after it is called out.

**Imprinter:** A device for marking characters onto a form. Imprinters include typewriters, printers, presses, pressure plates, and stamping machines.

**Increment:** Increase in value (usually) by adding one. Used a lot for counting (as in counting the number of repetitions through a loop).

**Index:** 1. A symbol or number used to identify a specific quantity in an array of similar items. 2. An ordered reference list of the contents of a file with keys for identification of the contents and the action required in preparing such a list. 3. In numerical control, movement of a machine part to a predetermined location.

**Information bits:** Any bits that are generated by the data source and not used for error control. Information theory provides that all information can be represented by some collection of bits,

regardless of the complexity of the information.

**Initialize:** To set to an initial or starting value; to do the preliminary steps that are not to be repeated, such as setting counters and addresses to zero. In ATARI BASIC, all nonarray variables are initialized to zero when the command run is given. Array and string elements are not initialized.

**Input:** 1. (noun) Information transferred to the computer. Output is information transferred away from the computer. In this manual, input and output are always in relation to the computer. 2. (verb) To transfer data from outside the computer (say, from a diskette file) into RAM. Output is the opposite, and the two words are often used together to describe data transfer operations: Input/Output or just "I/O". Note that the reference point is always the computer.

**Input device:** A device or set of devices used to bring data into another device. An input may be a channel for impressing or inserting a state or condition on another device, or a device or process involved in an input operation.

**Input/output:** Pertaining to either input or output or both. It is abbreviated I/O.

**Input/output channel:** A circuit path which allows independent communication between the processor and external devices. Input/output channels may transfer data between memory and external interfaces in blocks of any size without disturbing working registers in the processor. Multiple channels are allowed to operate concurrently with hardware priority control of each channel. Transfers are in full words, with automatic packing and unpacking allowed in some systems.

**Instruction:** A statement containing information which can be coded and used as a unit in a digital computer to command it to perform one or more operations. An instruction usually contains one or more addresses and may specify arithmetic operations such as addition or multiplication, or control operations for data manipulation. Instructions can be grouped as:

1. Data transfers between registers and memory.
2. Branching operations.
3. Input/output control.
4. Loading and storing accumulators.
5. Restoring registers and accumulators.
6. Jumps and stack-pointer operations.

## **Instruction—Keyboard**

7. Binary and decimal arithmetic.
8. Set and reset interrupts.
9. Increment and decrement registers and memory.

**Instruction code:** All of the symbols and definitions used to systematize the instructions for a given computer or executive routine.

**Integrated circuit:** An interconnected array of components, which is fabricated from a single crystal of semiconductor material by etching, doping, and diffusion, and which is capable of performing at least one and sometimes many complete circuit functions.

**Interactive:** A system that responds quickly to the user, usually within a second or two. All personal computer systems are interactive.

**Interface:** The electronics used to allow two devices to communicate. An interface enables devices to exchange information and implies a connection to complete the interchange.

**Interpreter:** An executive routine which translates a stored program written in a high-level language into machine code and performs the indicated operations using subroutines as they are translated.

**I/O:** Short for input/output, I/O devices include the keyboard, TV monitor, program recorder, printer, and disk drives.

**IOCB:** Input/Output Control Block. A block of data in RAM that tells the operating system the information it needs to know for an I/O operation. It is written as an arithmetic expression equal to a number between 1 and 7.

**I/O test program:** Refers to a special PROM containing a program which plugs into the checks input/output circuit boards.

**Item size:** The magnitude of an item expressed in words, characters, or blocks.

**K:** Stands for “kilo” meaning “times 1000”. Thus 1 Kbyte is (approximately) 1000 bytes. (Actually 1024 bytes.) Also, the device type code for the keyboard.

**Keyboard:** A unit containing keys for entering data or information into a system. Keyboards may be alphanumeric (as used for word processing, text processing, and data processing) or numeric as used for Touch-Tone telephones, accounting machines, and calculators.

**Keyword:** A word that has meaning as an instruction or command in a computer language, and thus must not be used as a variable name or at the beginning of a variable name.

**Kilobyte or K:** 1024 bytes of memory. Thus a 16K RAM capacity actually gives us  $16 \times 1024$  or 16,384 bytes.

**Language:** A set of conventions specifying how to tell a computer what to do. A language consists of a carefully defined set of characters and rules for combining the characters into larger units such as words or expressions. There are also rules for word arrangement and usage to achieve specific meanings. Most computer languages have the following features:

1. Data objects or structures with descriptions that correspond to nouns and adjectives in natural languages.

2. Operations and commands which act upon the data objects, corresponding to verbs and adverbs in a natural language.

3. Control structures to specify the sequence of operations which correspond to phrasing and forming paragraphs in natural languages.

**Language interpreter:** A processor, assembler, or other routine that accepts statements in one language and produces equivalent statements in another language.

**Language translator:** A program or routine that converts statements in one language to equivalent statements in another language. The languages may be computer or machine languages, or natural languages such as English.

**Least significant byte:** The byte in the rightmost position in a number or a word.

**Light pen:** A photosensitive device that can cause the computer to change or modify the display on a CRT screen or can be used to enter input into a program. As the display information is selected by the operator, the light pen signals the computer using a pulse. The computer can then instruct other points or lines to be plotted on the screen following the pen movements, or can utilize the input in the program.

**List:** An ordered set of items. A pushdown list is a set of items in which the last item entered is the first item in the list, and the position of the other items is pushed back by one. A pushup list has items entered at the end of the list, and the other items maintain the same relative position in the list.

## **Load—Matrix printer**

**Load:** The process of filling a storage unit in a computer. Loading includes the reading of the beginning of a program into storage and the modifications necessary to the program for the transfer of control for execution. Loading also includes the transfer of storage between memory units. A typical load operation transfers the contents of a memory byte and stores it in the accumulator. The memory is read bit by bit, and as each bit is read, the next sequential accumulator bit will be set or reset to reproduce the status of the memory bit just read.

**Logic gates:** A circuit or single component capable of performing a logic operation. A gate may have several inputs but only one output.

**Loop:** A self-contained series of instructions in which the final instruction can modify itself, causing the process to be repeated until a terminal condition is reached. Productive instructions in the loop are used to manipulate operands while housekeeping or bookkeeping instructions modify the productive instructions and keep track of the number of operations and repetitions. A loop may be terminated under any number of conditions.

**Machine:** A general term for a device such as a microprocessor or microcomputer that can store and process numeric and alphabetic information. Machine is used to refer to both analog and digital computers along with related data processing equipment.

**Machine-code:** An operation code that a machine is designed to recognize directly and without translation.

**Machine language:** A language that can be used directly by a microprocessor; a binary language. All other languages must be translated or compiled into binary code before entering the processor. Users generally write the program in coded instructions that are more meaningful to them. Then assembly programs are used to translate the symbolic instructions into binary machine code.

**Magnetic core storage:** A storage system that uses a core of magnetic material coupled by wires or other conductors. The cores are magnetized to represent binary ones or zeros.

**Matrix printer:** A device that uses an array of dots to form characters. Dot-matrix character formation is also used in some display devices. Sometimes called dot matrix printer.

**Memory:** The part of a computer (usually RAM or ROM) that stores data or information.

**Memory bus:** The circuit path for communication between the CPU and memory. A memory bus may actually consist of three buses which are time-shared over a single bus:

1. Memory address bus.
2. Memory-to-CPU data bus.
3. CPU-to-memory data bus.

**Memory scan:** An option which provides a rapid search of any part of memory for any word. With a memory-scan option, any block of locations may be searched using a single instruction.

**Menu:** A list of options from which the user may choose.

**Microcomputer:** A computer based on a microprocessor chip; in ATARI's case, the 6502. A microcomputer is a complete small computing system consisting of hardware and software. The hardware includes the microprocessing unit, memory, auxiliary circuits, power supply, and control panel. The main processing blocks are typically fabricated with large-scale integration circuit packages.

A typical microcomputer has the CPU on a single chip or circuit board; the system also contains ROM storage for programs and data along with clock circuits, input and output units, selector registers, and control circuits. The console control panel typically has a typewriter-type keyboard. The keyboard is an input port. Standard software often includes an assembler, loader, source editor, and debugging and diagnostic routines.

**Microprocessor:** The large-scale integration equivalent of a computer's central processing unit, designed to work as a sequential computational or control unit by executing a defined set of instructions contained in memory. In a microcomputer with a fixed instruction set, the microprocessor may consist of an arithmetic-logic unit and a control-logic unit. In a microcomputer with a microprogrammed logic set, it may contain an additional control-memory unit. The microprocessor determines what devices should have access to data and makes these decisions based upon timing requirements. It also correlates the activities of the memory and input/output units.

All microprocessors use large-scale integrated circuit technology; some feature a single-chip construction while others use

## **Microprocessor—Monitor system**

several chips. The division or partitioning of functions is based on considerations of word length, flexibility, and performance.

**Microprogramming:** The techniques of modifying an instruction set by building higher-level instructions from basic elemental operations. The higher-level instructions can then be directly programmed. If a machine has basic instructions for addition, subtraction, and multiplication, an instruction for division could be defined by microprogramming. Microprogramming adds flexibility to the microcomputer. Compared to conventional programming, the distinctive feature is the storage associated with the control unit. In microprogramming, a user instruction determines an address in control memory which provides the starting point for executing the microprogrammed steps.

Microprogramming enhances flexibility by allowing the machine to optimize its control memory for specific applications. This ability to adjust can greatly simplify programming. It is generally slow, since each user instruction requires that a sequence of programmed steps be executed. Control storage ROM must be fast enough to allow the use of a separate clock that is five to ten times faster than that used for main memory.

**Mnemonic:** 1. A word-like symbol used in assembly languages to cause a computer to execute an instruction whose description resembles the word symbol. The term MOV A, B is a mnemonic for “move the contents of register B into arithmetic register A.”

**Modem:** An electronic device that performs the modulation and demodulation functions required for communications. A modem (formed from a blending of modulator and demodulator) can be used to connect computers and terminals over telephone circuits. On the transmission end, the modulator converts the signals to the correct codes for transmission over the communications line. At the receiving end, the demodulator reconverts the signals for communication to the computer using the computer interface unit.

**Monitor:** The television receiver used to display computer output.

**Monitor system:** The hardware and software used to control computer system functions. The monitor system can simulate the processor, maintain continuity between jobs, observe and report on the status of input/output devices, and provide automatic accounting of jobs.



**Most Significant Byte:** The byte in the leftmost position in a number or a word.

**Multiplier:** A device that generates a product from two numbers. A digital multiplier generates the product from two digital numbers by additions of the multiplicand in accordance with the value of the digits in the multiplier. It then shifts the multiplicand and adds it to the product if the multiplier digit is a one, or shifts without adding if the digit is a zero. This is done for each successive digit of the multiplier.

**NOR:** A logical operation that has a true output only if all inputs are zero (false). The negative-OR operation.

**NOT:** A logic operator having the property: if P is a statement, then the NOT of P is true if P is false; and it is false if P is true. The NOT operator is represented by an overline.

**NOT AND:** The NAND operation.

**NOT AND gate:** An AND gating circuit combined with an inverter; a NAND gate.

**Notation:** The act, process, or method used to represent technical facts or quantities. In computer practice, the term typically describes the number radix, as follows:

NOTATION	RADIX
BINARY	2
TERNARY	3
QUATERNARY	4
QUINARY	5
DECIMAL	10
DUODECIMAL	12
HEXADECIMAL	16
DUOTRICENARY	32
BIQUINARY	2,5

**NOT if-then gate:** A gate that performs the A AND NOT B and B AND NOT A operations.

**Null String:** A string consisting of no character whatever. For example, A\$="" stores the null string as A\$.

**Numerical control:** The automatic control of a machine or process using numerical data that is introduced using punched tapes or other input methods. Most numerical-control devices have limited logical capability, and they rely on the input medium for

## **Numerical control—Operator console**

---

detailed guidance. A computer is generally used to prepare the control media.

**Object code:** Machine language derived from "source code," typically from Assembly Language.

**Octal:** The octal numbering system uses the digits 0 through 7. Address and byte values are sometimes given in octal form.

**One-address:** A system of instructions such that each complete instruction explicitly describes one operation and one storage location.

**Open:** To prepare a file for access by specifying whether an input or output operation will be conducted, and specifying the filespec.

**Operand:** 1. The fundamental quantity on which a mathematical operation is performed. A statement usually consists of an operator and an operand; in an add instruction, the operator will indicate "add," and the operand will indicate what is to be added. 2. A result, parameter, or the address portion of an instruction.

**Operating console:** A unit which contains all controls and indicators necessary for the operation of the processor. A typical microcomputer operating console may include the following functions:

1. Run indicator and switch
2. Halt indicator and switch
3. Reset switch
4. Link indicator
5. Interrupt indicator
6. Accumulator program counter, and memory display

**Operation code:** A code that represents specific actions. Also called instruction code.

**Operation decoder:** A device that selects one or more channels of operation according to the operator part of the machine instruction.

**Operator:** 1. The mathematical symbol which represents the process to be performed on an associated operand. 2. The portion of an instruction which indicates the action to be performed on the operands.

**Operator console:** A hardware unit which allows the operator to communicate with the computer. The operator console is used to enter data or information, to request and display stored data, and to actuate the various command routines.

**OR:** A logic operator having the following property for logical quantities P and Q:

P	Q	P OR Q
0	0	0
0	1	1
1	0	1
1	1	1

The OR operator is represented in electrical terminology by a “plus” symbol.

**OR ELSE:** A logical operator which states that if P and Q are statements, then P OR ELSE Q is either true or false. Also called EITHER/OR.

**OS:** Abbreviation for operating system. This is actually a collection of programs to aid the user in controlling the computer.

**Output:** The produced signals used to drive peripheral terminals. See I/O.

**Paging:** The procedure used to locate and transmit pages between main storage and auxiliary storage, or to exchange them with pages of the same program or other programs. Paging can be used to assist in the allocation of a limited amount of main storage among several concurrent programs.

**Parallel:** Two or more things happening simultaneously. A parallel interface, for example, controls a number of distinct electrical signals at the same time. Opposite of serial.

**Pen light:** A pen-like device with light and a photosensor on one end for communicating with a computer through a CRT device.

**Peripheral:** An I/O device. See I/O.

**Permutation:** Any one of the total number of changes in position or form that are possible in a group.

**Pilot system:** A collection of file records and data obtained from a business over a period of time and used for simulation purposes.

**Pip:** A spot of light on a CRT screen for display pointing, calibration.

**Pixel:** Picture Element. One point on the screen display. Size depends on graphics mode being used.

**Precedence:** Rules that determine the priority in which operations are conducted, especially with regard to the arithmetical/logical operators.

## **Problem language—Programming**

**Problem language:** The language used by a programmer in stating the definition of a problem.

**Processor:** A hardware data processor or a program that performs the functions of compiling, assembling, and translating for a specific language. Processor operations can involve registers, accumulators, program counters and stacks, and input/output and internal instruction control.

**Processor module:** The circuit card that contains the microprocessor along with the logic and control circuitry necessary to operate as a processing unit. The processor support circuits consist of clock circuits, multiplexer, bus control, input/output control, and interrupt logic.

**Program:** 1. A set of instructions arranged in the proper sequence for directing the computer in performing a desired operation, such as the solution of a mathematical problem or the sorting of data. 2. To prepare a set of ordered instructions for automatic computer operation.

A program includes plans for the transcription of data, coding for the computer, and plans for the absorption of results into the system.

**Program counter:** A counter which contains the address of the next instruction to be fetched from memory. The address is automatically incremented after an instruction fetch except when modified by branch instructions. During interrupts, the program counter saves the address of the instruction.

**Program error:** A mistake made in the program code by the programmer, keypuncher, compiler, or assembler.

**Program library:** A collection of available computer programs and routines. A typical microcomputer library might include a text editor, loader, assembler, RAM test program, decimal addition, PROM programmer, A/D conversion, logic subroutines, and BCD-to-binary conversion.

**Programmable logic:** Devices and systems which provide logic functions that can be changed by the user. Programmable logic devices include FPLAs, PLAs, ROMs, EAROMs, RAMs, CAMs, and microprocessors. Programmable logic systems include microcomputers, programmable calculators, minicomputers, and large-scale computers.

**Programming:** The design, writing, and testing of programs. Programming may consist of:

1. Definition of problem.
2. Preparation of flowchart.
3. Listing of computer instructions.
4. Selecting circuit patterns or control modes.

**Programming flowchart:** A flowchart used to represent the sequence of operations in a program.

**Prompt:** A symbol that appears on the monitor screen that indicates the computer is ready to accept keyboard input. In ATARI BASIC, this takes the form of the word "READY". A "?" is also used to prompt a user to enter (input) information or take other appropriate action.

**Pseudo code:** A code that requires translation prior to execution. Pseudo codes are often used to link a subroutine into the main routine, and they usually express programs in terms of source language by referring to locations and operations using symbolic names and addresses.

**Pseudo instruction:** A group of characters having the same general form as an instruction, but never executed as an actual instruction. Pseudo instructions are used as symbolic representations in compilers, interpreters, and assemblers to designate groups of instructions for performing a particular task.

**RAM:** Abbreviation for random-access memory. The main memory in most computers. RAM is used to store both programs and data. It is a mass store that provides fast access to any storage location point. A RAM system can be divided into three main sections:

1. Address buffers, read-write logic, and chip-select logic.
2. Data bus buffers and memory array.
3. Refresh and control logic.

During a cycle, a 1 on a write input/output line will be interpreted by the RAM as a write enable command, and data on the bus will be written in. Available RAMS have access times from 2 nanoseconds to 3 seconds and range in size from 4 to 4096 bits per chip.

**Random number generator:** May be hardware (as is ATARI's) or a program that provides a number whose value is difficult to predict. Used primarily for decision-making in game programs, etc.

## **Record—Selector**

**Record:** A block of data.

**Reserved word:** See Keyword.

**Reset:** To return a device such as a register to zero or to an initial or selected condition.

**Restore:** To return an index, address, word, or character to its initial state, sometimes by periodic recharge or regeneration.

**ROM:** Abbreviation for read-only memory. In this type of solid-state electronic memory, information is stored by the manufacturer and it cannot be changed by the user. Programs such as the BASIC interpreter and other cartridges used with the ATARI systems use ROM.

**Routine:** An ordered set of instructions used to direct the computer to perform one or more specific tasks or operations. A closed routine is one that is not entered as a block of instructions within the main routine, but entered by a linkage from the main routine. A routine may be considered as a subdivision of a program with two or more instructions that are functionally related.

**Run:** The execution of a program on or of several routines linked together so that they form an operating unit. Usually during a run, manual operations are minimal; a typical run may involve loading, reading, processing, and writing.

**Save:** To copy a program or data into some location other than RAM (for example, diskette or tape).

**Screen:** The TV screen, In ATARI BASIC, a particular I/O device codes "S:"

**Sector:** The smallest block of data that can be written to a disk file or read from one. Sectors can store up to 128 bytes (single density) or 256 bytes (double density).

**Segment:** 1. To divide a routine into parts with each one capable of being completely stored in internal storage and containing the necessary instructions to jump to other segments. 2. A part of a routine short enough to be sorted entirely in the internal storage of a computer, yet containing the coding necessary to call in and jump to other segments. A segment can be placed anywhere in memory and addressed relative to a common origin.

**Selector:** 1. A switching operation based on previous processing which allows a logical choice to be made in the program or system. 2. A mechanical multiposition switch. 3. Cursor.

**Separator:** A specific character used to mark the logical boundary between items that can be considered as separate and distinct units.

**Serial:** The opposite of parallel. Things happening only one at a time in sequence. Example: A serial interface.

**Shift instruction:** A computer instruction which causes the contents of an accumulator register to shift to the right or left. A right shift instruction is equivalent to dividing by two; a left shift instruction is equivalent to multiplying by two, or adding the contents of the register to itself.

**Signal:** The intelligence, message, or effect to be conveyed over a transferal system; it may be electrical, visual, or audible. It is the event or phenomenon for conveying the data from one point to another.

**Software:** As opposed to Hardware. Refers to programs and data.

**Software package:** The programs or sets of programs used in a particular application or function. Many software packages include diagnostic programs for verifying processor and memory operation along with cross-assemblers.

**Software system:** The entire set of programs and software development aids used in a microcomputer system.

**Source:** The device or address that contains the data to be sent to a destination.

**Source code:** A series of instructions, written in language other than machine language, which requires translation in order to be executed.

**Special character:** A character that can be displayed by a computer but is neither a letter nor a numeral. The ATARI graphics symbols are special characters. So are punctuation marks, etc.

**Specific routine:** A routine used to solve a particular arithmetic, logic, or data-handling problem in which each address refers to explicitly stated registers and locations.

**Statement:** An instruction to the computer. See also Command. While all commands may be considered statements, all statements are certainly not commands. A statement contains a line number (deferred mode), a keyword, the value to be operated on, and the RETURN command.

**Storage:** A device or medium on or into which data can be entered, held, and retrieved later; a memory. Storage may use electro-

## **Storage—Track**

static, magnetic, acoustic, optical, electronic, or mechanical methods.

**String:** (1) A sequence of letters, numerals and characters which may be stored in a string variable. The string variable's name must end with a \$. (2) Any linear sequence of items which are grouped in a series according to certain rules. A string may be a set of records grouped in ascending or descending sequence according to a key contained in the records.

**Subroutine:** A part of a program that can be executed by a special statement (gosub) in BASIC. A subroutine can be a portion of another routine which allows the computer to carry out a defined mathematical or logical operation. This effectively gives a single statement the power of a whole program. The subroutine is a very powerful construct.

**System data bus:** A bus designed for communications between all external units and the CPU. A typical bus transfers information between any two devices connected to the bus by granting access through a priority network for bus control.

**System handbook:** A document that provides a concise reference of all the major characteristics of the microcomputer system. The system handbook will contain operation codes, addressing modes, service status details, interfaces, timing diagrams, and data flow.

**Termination:** The end of a program or program run.

**Text editing:** An editing facility designed into a text-editing program to allow the keyboard entry of text or copy without regard to the final format. After the copy has been placed in storage, it can be edited and justified by specifying the desired format.

**Tokenizing:** The process of interpreting textual BASIC source code and converting it to the internal format used by the BASIC interpreter.

**Trace routine:** A routine used to provide a time history of machine registers and controls during the execution of the object routine. A complete tracing routine can provide the status of all registers and locations affected by each instruction each time the instruction is executed.

**Track:** A circle on a diskette used for magnetic storage of data. Each track has 18 sectors, each with 128 byte storage capability (single density). There are a total of 40 tracks on each diskette.



**Variable:** A variable may be thought of as a box in which a value may be stored. Such values are typically numbers and strings.

**Visual display terminal:** This is a device that permits inputs to a computer system through a keyboard or other manual terminal or input device such as a light pen, and whose primary output is visual through a CRT unit or other type of display. The terminals allow keyboarding, verification, editing, correction, and reformatting of material.

**Window:** A portion of the TV display devoted to a specific purpose such as for graphics or text.

**Word processing:** The preparation of printed material using automatic data processing equipment.

**Write-Protect:** A method of preventing the disk drive from writing on a diskette. ATARI diskettes are write-protected by covering a notch on the diskette cover with a small sticker.

**Zone Bit:** A bit in a group of positions used to indicate a specific class of items or, in some cases, a bit used as a key to a code.

# Index

## A

Access time, 28  
Addition, binary, 31-35  
Addresses, 81  
Algebra, Boolean, 44-79  
Analog computers, 12  
AND (Boolean operation), 50, 51  
AND (Boolean term), 46, 47  
Arithmetic, computer, 17, 31-43  
Arithmetic functions, 14, 15, 143, 144  
Arithmetic-logic unit, 16  
Assembly language programming, 199, 200  
ATASCII character set, 229

## B

BASIC, 81, 82, 107  
BASIC commands, 127-131  
BASIC language cartridge, 107  
BASIC programming statements, 87-96  
BASIC reserved words, directory, 220-225  
BCD code, 11  
Binary arithmetic, 31-43  
Binary coded decimal, 10  
Binary codes, 10  
Binary digit, 4  
Binary number codes, 198  
Binary numbers, 1  
Binary number system, 3, 4  
Bits, 3  
Boolean algebra applications to switching circuits, 49-58  
Boolean algebra laws, 56-59  
Boolean classes, 46, 47  
Boolean element, 46, 47  
Boolean equation mechanization, 59-79  
Boolean equation simplification, 59-79  
Boolean expressions, relational, 44, 45  
Boolean operations, 44-79  
Boolean symbols, 47-49  
Bus, address, 27  
Bus, computer, 26, 27  
Bus, control, 27  
Bus, data, 27, 28  
Bus systems, 26, 27

## C

Cartridges, program, 98, 99  
Cassette input /output, 140, 141  
Cassette recorder, 105, 106, 137

Cassette recorder, loading programs from the, 118  
Cassette recorder, saving programs on the, 116-118  
Cathode ray tube, 17  
Celsius-Fahrenheit temperature conversion, 83-88  
Central processing unit, 16  
Checkbook balancer program, 161-164  
Chip, 25, 26  
Chips, 12  
Clock, master, 28, 29  
Codes, 1  
Commands, graphic, 174-192  
Commands, input /output, 138-142  
Commands, sound, 192, 193  
Complements, Boolean, 193  
Computer blues program, 195  
Computer components, 97-106  
Computer elements, basic, 15-17  
Computers, analog, 12  
Computers, digital, 12, 15  
Computer structure, 15-17  
Continuing education, 205-219  
Control, 16, 17, 19, 20  
Control characters, printed versions, 238  
Controllers, 121, 122  
Conversions between number systems, BCD to decimal, 11  
Conversions between number systems, binary to decimal, 7  
Conversions between number systems, binary to hexadecimal, 9, 10  
Conversions between number systems, binary to octal, 7  
Conversions between number systems, decimal to binary, 4-7  
Conversions between number systems, decimal to binary decimals, 40-43  
Conversions between number systems, decimal to octal, 8  
Conversions between number systems, octal to decimal, 8  
CPU, 25  
Currency exchange program, 149-151

## D

Data manipulation, 19-23  
Data storage, 20, 21  
Decimal number system, 1-3

DeMorgan's Theorem, 76, 79  
 Derived functions, 237  
 Digital computers, 12, 15  
 Direct memory access, 26  
 Disk drives, 122-134, 137, 138  
 Diskettes, 123  
 Disk operating system (DOS), 124-127  
 Disk operations, 127-134  
 Disk operations, BASIC commands used with, 127-131  
 Disk utility package (DUP), 124, 125  
 Division, binary, 39, 40

## E

Edit features, 102-105  
 Education, computer, 205-219  
 Elements, interconnection, 18  
 Error messages, BASIC, 131-134  
 Exclusive OR, 56

## F

File management subsystem (FMS), 126, 127  
 File names, 126  
 Flowcharts, 84-86, 165-173  
 Fractions, binary, 40-43  
 Functions, arithmetic, 143, 144  
 Functions, computer, 12, 13  
 Functions, derived, 237  
 Functions, secretarial, 13  
 Functions, special purpose, 145, 146  
 Functions, trigonometric, 144

## G

Glossary, 242  
 Graphics, 174-192  
 Graphics commands, 174-186  
 Graphics modes, 174-177  
 Graphics modes, color, 176, 177  
 Graphics window, 177  
 Greatest common divisor program, 157-159  
 Gun collector program, 159-161

## H

Hardware, 15  
 Hexadecimal numbers, 1  
 Hexadecimal number system, 8-10  
 Hexcode loader program, 200  
 High-level language, 81

## I

IC, 25, 26  
 ICs, dedicated, 26  
 Input, 16  
 Input/output commands, 138-142  
 Input/output devices, 137, 138

Inputs, computer, 15, 16  
 Instructions, arithmetic-logic, 18  
 Instructions, control, 18  
 Instructions, data movement, 18  
 Instructions, input/output, 18  
 Integer, 2  
 Instruction set, 18  
 Integrated circuits, 25, 26  
 Interconnection of computer elements, 18  
 Interface, RS-232, 138  
 IOCB (input/output control block), 137  
 I/O operations, 135-142

## J

Joystick, 122

## K

Keyboard, 99-105, 137  
 Keyboard input/output, 141  
 Keys, special, 99-105

## L

Large-scale computers, 23, 24  
 Laws of Boolean algebra, 56-59  
 Learning to program, 108-116  
 Lessons, correspondence, 208-219  
 Light show program, 188  
 Line numbers, 88, 89

## M

Machine language, 198-204  
 Margins, 175  
 Master bus control, 27  
 Master element, 27  
 Mathematics, binary, 31-43  
 Maxterms, Boolean, 47  
 McGraw-Hill Continuing Education Center, 205  
 Medium-scale computers, 24  
 Memory, computer, 14, 16  
 Memory locations, 239  
 Memory map, 234-236  
 Memory modules, 97, 120  
 Memory saving techniques, 147, 148  
 Metric conversion program, 151-156  
 Microcomputer basics, 12  
 Microcomputers, 24, 25  
 Microcomputers, one chip, 26  
 Microcomputer terms, 242  
 Microprocessors, 23, 25, 26  
 Minicomputers, 24  
 Minterms, Boolean, 47  
 Multiplication, binary, 37-39  
 Multiply instruction, 18

## N

NAND (Boolean operation), 55  
National Technical Schools, 205, 207  
NOR (Boolean operation), 53-55  
NRI (correspondence school), 205  
Number systems, 1  
Number systems, binary, 3, 4  
Number systems, decimal, 1-3  
Number systems, hexadecimal, 8-10  
Number systems, octal, 7, 8  
Number systems and codes, 1

## O

Octal numbers, 1  
Octal number system, 7, 8  
OR (Boolean operation), 52  
OR (Boolean term), 47  
Output devices, 15  
Outputs, computer, 17

## P

Paddle, 121, 122  
Pathways, bidirectional, 19  
Pathways, data, 19, 23, 26, 27  
Peripherals, 97, 106, 119-134  
Printer, 137  
Printer input/output, 141, 142  
Printers, 119, 120  
Program, checkbook balancer, 161-164  
Program, computer blues, 195  
Program, currency exchange, 149-151  
Program, greatest common divisor, 157-159  
Program, gun collector, 159-161  
Program, hexcode loader, 200  
Program, light show, 188  
Program, metric conversion, 151-156  
Program, text modes character print, 186  
Program, type a tune, 193  
Program, United States flag, 189  
Program, video graffiti, 190  
Programming, 80-96  
Programming, purposes for, 106  
Programming, simple steps in, 108-116  
Programming in machine-language, 198-204  
Programming objectives, 82  
Programming statements, BASIC, 87, 90, 96  
Programming steps, 82-88  
Program recorder, 105, 106, 137  
Programs, computer, 13, 15  
Programs, sample, 149-156

Pseudocode, 173

## R

Radix, 1  
RAM, 30, 107  
Random-access memory, 30  
Read command, 20, 21  
Read-only memory, 30  
Register, 20  
Registers, storage, 28  
ROM, 30, 98, 107  
RS-232 interface, 138

## S

Saving programs on tape, 116-118  
Schools, correspondence, 205-208  
Screen editor, 138  
Screen output, 142  
Size, computer, 23-26  
Slave element, 27  
Software, 15, 17, 18  
Sound, 192-197  
Sound commands, 192, 193  
Special purpose functions, 145, 146  
Storage, computer, 17  
Storage types, kinds of, 30  
Subtraction, binary, 36, 37  
Symbology, Boolean, 47-49

## T

Techniques, flowcharting, 165-173  
Temperature conversion program, 83-88  
Text modes character print program, 186  
Text window, 177  
Transistors, 12  
Trigonometric functions, 144  
TV monitor, 138  
Type a tune program, 193

## U

United States flag program, 189  
USR function, 198

## V

Veitch diagrams, 61-76  
Venn diagrams, 47, 49-56  
Video graffiti program, 190

## W

Write command, 20, 21  
Writing to storage, 27, 28

## X

XOR (Boolean operation), 56



ISBN 0-8306-0453-7