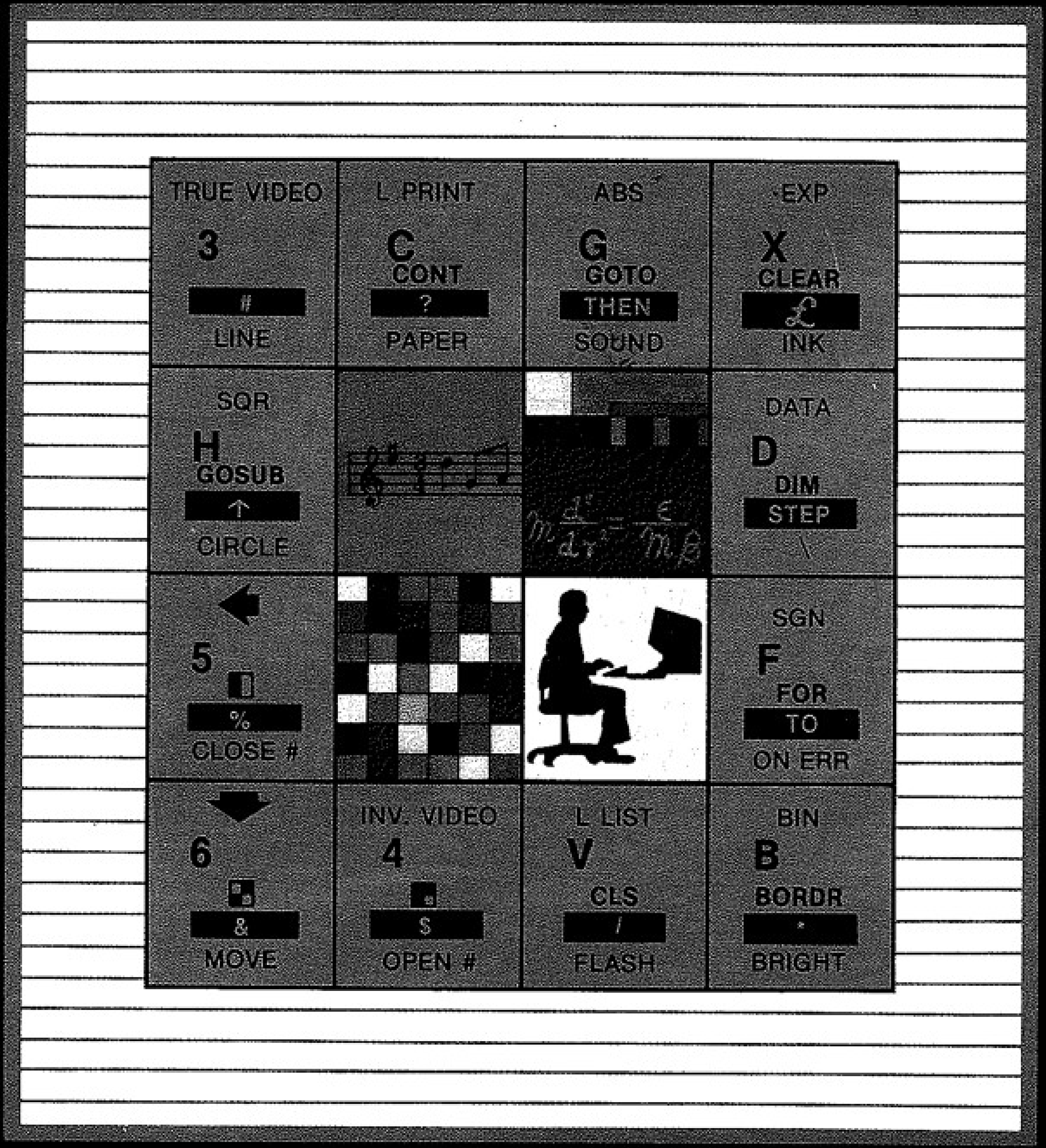
SAMS

22226

Timex® Sinclair 2068 Intermediate/Advanced Guide

Jeff Mazur



Timex Sinclair
2068
Intermediate/Advanced
Guide

Jeff Mazur has been an electronics hobbyist since he was seven years old. In the past ten years, microcomputers have taken a dominant role in his leisure activity. He bought one of the first Apple computers and then began developing peripherals to enhance its operation. This became the basis for a small company that he founded to design and manufacture accessories for personal computers.

Besides tinkering with computers, the author is also an active amateur radio operator and photographer and has written a monthly column for *Softalk* magazine. Mr. Mazur holds a degree in physics from UCLA and lives in southern California where he works as a videotape engineer for a major television network.

Timex Sinclair 2068 Intermediate/ Advanced Guide

by

Jeffrey Mazur

Howard W. Sams & Co., Inc.
A Publishing Subsidiary of ITT
4300 West 62nd Street, Indianapolis, Indiana 46268

FIRST EDITION
FIRST PRINTING — 1984

International Standard Book Number: 0-672-22226-4
Library of Congress Catalog Card Number: 84-50069

Edited by *Jim Rounds*
Illustrated by *Jill E. Martin*

Printed in the United States of America.

Preface

This is the second of two books on the T/S 2068 computer. The first book, *Timex Sinclair 2068 Beginner/Intermediate Guide,* is a guide to getting started with the T/S 2068 computer. It deals with setting up and operating the computer, and learning to program in BASIC. This second book will take you beyond BASIC — into the world of computer circuits, the Z80 microprocessor, the BASIC interpreter, and machine language programming. The main goal is to show you how to make your programs run faster and do things that are otherwise impossible with BASIC. At the same time, you will learn a lot more about how your computer works.

This book will assume only a working knowledge of the T/S 2068 and BASIC programming. That is, you should know how to turn on the computer and be able to enter and run simple BASIC programs. You *do not* need to know anything about bits and bytes or RAM and ROM — it begins with a simple explanation of what a computer is and how it works. If you are new to computers, or are not familiar with BASIC programming, then you may wish to start with the *Timex Sinclair 2068 Beginner/Intermediate Guide.*

This book serves two purposes. It describes how a computer works, and it introduces the reader to machine language programming. The first part of the book covers computer basics: the operation of computers in general, and the T/S 2068 in particular. It includes a description of most of the T/S 2068 *hardware.* The second part introduces *machine language* — the native tongue of the Z80 microprocessor that drives the T/S 2068. Part Two ends with a section that describes how machine language routines can be used from within BASIC programs.

This book also serves as a reference guide for the T/S 2068 computer. It contains complete information on the various expansion connectors for the hardware buff trying to interface a new device to the computer. The machine language programmer will find a wealth of technical information on the Z80 Central Processing Unit and the AY-3-8912 Programmable Sound Generator used in the T/S 2068. This book describes *in detail* each instruction in the Z80's repertoire. In addition to the hardware descriptions, the book also discusses the inner workings of the BASIC interpreter, the special memory mapping scheme used by the T/S 2068, and the Function Dispatcher routines which make up the heart of the computer's *operating system.* Indeed, there is something for everyone.

Hopefully this book will widen your horizons and unravel a few of the mysteries that surround computers. When you've finished reading it, you will want to keep it near your computer so the reference material is readily available.

JEFFREY MAZUR

I gratefully acknowledge all of the help and support I received from my wife Lynne, and from my children, Michael, Jonathan, and Jessica, who put up with "daddy's book."

Contents

PART 1—How Computers Work

Section A—Computer Basics

CHAPTER 1

CHAPTER 2

CHAPTER 3

Section B—Inside the T/S 2068

CHAPTER 4

CHAPTER 5

CHAPTER 6

CHAPTER 7

CHAPTER 8

PART 2—Machine Language

Section A—Programming the Z80

CHAPTER 9

CHAPTER 10

CHAPTER 11

CHAPTER 12

CHAPTER 13

Section B—Machine Language on the T/S 2068

CHAPTER 14

CHAPTER 15

APPENDIX A

APPENDIX B

APPENDIX C

PART 1
HOW COMPUTERS WORK

SECTION A
COMPUTER BASICS

1
What Is a Computer?

INTRODUCTION

In the world of computers, the T/S 2068 stacks up as a relatively small, slow, and limited machine. It is considered a *personal* computer because it is used by only one person at a time, as opposed to larger computers that can handle many users simultaneously. The latter are referred to as *mainframe* computers or *minicomputers* depending on their size. Personal computers such as the T/S 2068 are also called *microcomputers*—primarily because they are built around a special electronic part or "chip" called a *microprocessor*. The T/S 2068 uses the *Z80*, one of the most popular microprocessors ever developed. Much of this book is devoted to describing how this tiny chip of silicon works.

Despite the many differences between a T/S 2068 and an IBM 370 mainframe computer, they both share a common design philosophy. In fact, it would be safe to say that almost all present-day computers fit the description of a *stored-program, digital, electronic, data processing machine*. This rather lengthy description also sums up the history of computing machines.

THE COMPUTER AS A MACHINE

Above all, the computer is a *machine*: it allows us to perform some function easier or faster than we can manually. It also allows us to accomplish some tasks that are otherwise impossible. As an example, consider another machine that you may be more familiar with — the *lever*. Long ago it was learned that a lever could be used to move objects that were too heavy to move by hand. The lever accomplishes this task by transforming a small force applied over a large distance into a larger force applied over a small distance. While levers help accomplish *physical* work, computers aid in the performance of *mental* work, otherwise known as *information processing* or *data processing*.

Mechanical vs. Electronic Machines

A lever is classified as a *mechanical* machine since it requires movement to perform its task. The first "mental machines" were also mechanical in nature. One of the first was the abacus, a mechanical machine that aids in the performance of a mental process: calculating numbers. Subsequently, other mechanical calculators were developed, including slide rules and adding machines.

Today, however, most of our mental machines, including the modern computer, are *electronic* machines. They use stationary electronic devices instead of moving mechanical parts. It is the *control* of currents that is important in electronics. Therefore, the electronic analogy of the lever is called an *amplifier*. It is a device that takes a small electric current (or signal) and uses it to control a larger one.

Depending upon the arrangement of other components in an electronic circuit, amplifiers can accomplish tasks such as inverting, adding, performing logarithms, or storing electronic signals. Therefore, almost every electronic circuit is based upon the use of amplifiers. Consequently, much of the relatively brief history of electronics has involved finding the best electronic amplifying device. The original amplifiers were vacuum tubes. Today they have been replaced by solid state transistors.

Analog vs. Digital Computers

Early in their history, computers dealt with data as voltage signals in a one-to-one relationship. If a signal level of 2 volts represented 200 miles per hour (mph), for example, then 1 volt represented 100 mph, and 0.52 volts was 52 mph, etc. Information treated this way is referred to as *analog*, thus early computers were called *analog computers*.

When analog computers had their day (it was a relatively short one), they performed calculations in seconds that would have taken many hours to perform by hand, even with a slide rule. But such computers were limited to dealing with *numbers*—for example, calculating the trajectory of a rocket. It was soon discovered, however, that almost any type of information could be processed if it was in *digital* form, that is, in a series of discrete chunks. This was a major breakthrough in the design of computers.

To appreciate the difference between analog and digital forms of information, consider these few examples. Analog information can have an infinite number of values: distance, time, and temperature, are all examples. We can have a stick that is 4 feet long, 4.527 feet long, or any possible value in between. Digital information, however, is restricted to discrete values: the flip of a coin is either heads or tails, the change in our pocket consists of a finite set of coins, and we can only have whole numbers of children (census figures notwithstanding, no family can have 2.3 children).

In a *digital computer*, all information to be processed (whether it represents numbers, text, or electronic signals) is first converted to a series of signals represented by binary ones and zeros. Dealing with only digital signals, the electronics are much simpler. When using amplifiers, for example, you need not worry about nonlinearities or distortion, since they operate either fully on or fully off. In this sense, the amplifier (tube or transistor) becomes an electronic switch. This gives rise to some specialized circuits called *gates*. The name is aptly chosen since electric current moves through these circuits only when the "gate" is open. We'll discuss gates in detail a little later. The point here is that gates, which form the basis of digital processing, are very easy to make. Today, in fact, thousands of gates can be squeezed onto a piece of silicon no larger than a pencil eraser. The development of the modern computer, then, was from the mechanical to the electronic, and from the analog to the digital.

Stored Program Architecture

While digital electronics gave a great boost to computer design, a major stumbling block still remained. The early computers were programmed by connecting wires to various circuit elements. Making a simple change in the program often involved a complicated and time-consuming process. This cumbersome technique also limited the program size. The idea of storing the program itself within the machine's memory was the final link leading to today's computers. By replacing the wires with a *bit pattern* in memory, programs were made more flexible and were limited in size only by the amount of memory available. This practice, known as the *stored-program* technique, was a milestone in computer science—only today have we begun to reach its greatest potential.

Stored-program machines operate by executing instruction cycles. Each cycle begins by having the computer fetch an instruction from memory. This information is then decoded by the computer to direct its further action. In most cases, the computer will read one or two more pieces of information to finish its cycle. All of this takes place extremely fast—in the T/S 2068, up to one million instructions can be executed in one second!

THE FOUR BASIC COMPUTER PARTS

Any computer can be divided into four main parts: the input section, the Central Processing Unit, the memory, and the output section. Fig. 1–1 shows how these sections are connected. The arrows indicate which direction information can flow.

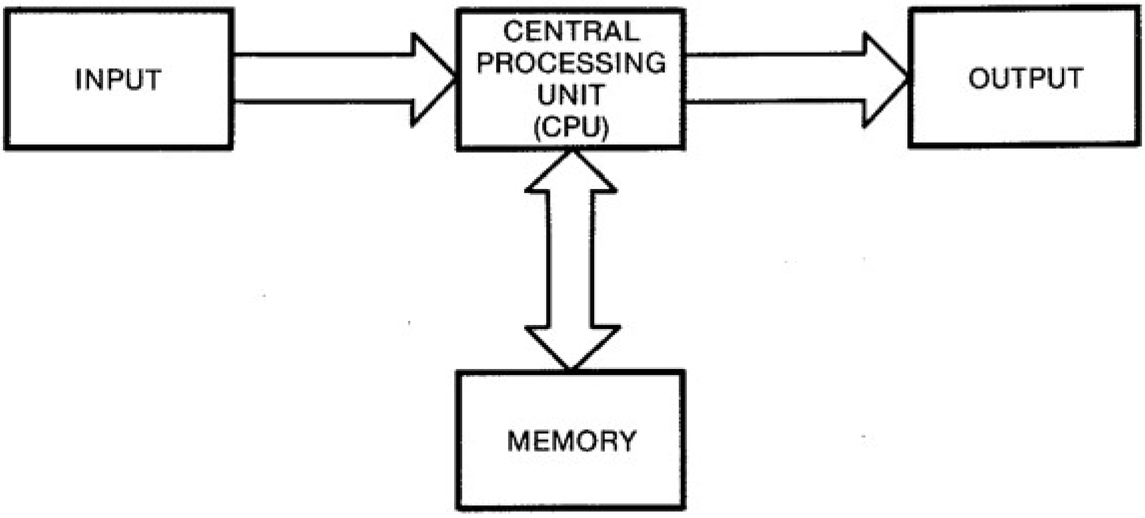


Fig. 1–1. The four basic parts of a computer.

Input

The *input* section is the connection to the outside world through which information is fed *into* the computer. In the case of the T/S 2068, there are three input devices supported by the standard machine. They are the keyboard, the joystick, and the cassette playback. Each of these devices allows the user to put information into the computer. Any connection between different parts of a computer, or between the computer and the human user, is called an *interface*. The keyboard, of course, is the main interface used for communicating with the computer and telling it what to do.

The CPU

The CPU, or Central Processing Unit, of the T/S 2068 is the Z80 microprocessor (described in detail in Chapter 4). The Z80 is only one of many microprocessors. For example, another popular one is the 6502 used in Apple, Atari, and Commodore computers. Each microprocessor has its own *architecture* and *instruction set*. These determine the *machine language* with which the CPU is programmed.

Memory

The CPU can only handle a small packet of information at one time. In the T/S 2068, the Z80 processes eight tiny pieces of information, called *bits*, in each packet. These packets, in turn, are called *bytes*. We'll have a lot more to say about bits and bytes later, but in general, you can consider each byte as representing a single character (letter, number, punctuation mark, etc.).

Since we usually want to process more than one byte of information—(the test scores of a class might be hundreds of bytes; a business letter, thousands)—we need some place to store this information, also called *data*, where the CPU can easily reach it. We also need storage for the information that tells the computer what to do, its *program*. Both data and program can be stored in a portion of the computer called *memory*. The computer's memory is divided into two major classifications called *RAM* and *ROM*.

RAM—There are many types of memory devices that can store information in a computer. Most of the devices accept data from the computer—they are "written to." They then hold the data indefinitely and feed it back when asked—they are "read out." This type of memory is referred to as Random Access Memory, or RAM for short. The T/S 2068 has 49,152 bytes or 48K internal RAM.*

ROM—Another type of memory device used in the T/S 2068 is called Read Only Memory, or ROM. This device is "written to" at the

*The "K" stands for kilobytes or a thousand bytes, from the metric prefix "kilo" meaning thousand. However, in the computer's binary number system, the closest "round" number to one thousand is actually 1024. Thus each "K" actually stands for 1024 bytes.

factory and can only be read by the computer. ROM is used to hold information that is essential to the operation of the computer, but which will not require any changes. The operating system and BASIC interpreter of the T/S 2068 are examples of information that is stored in ROM. An advantage of storing information in ROM is that the data is permanently available, even after the computer has been turned off and on again. In contrast, all information stored in RAM is lost whenever the power is removed.

Because of the preceding distinction, ROM is often referred to as *nonvolatile storage* and the type of RAM used in the T/S 2068 is considered *volatile storage*. The T/S 2068 contains 24K of internal ROM. Additional ROM and/or RAM can be added to the T/S 2068 through the front panel DOCK connector or through the expansion connector on the rear of the computer. See Chapter 5 for a complete description of the T/S 2068's memory capabilities.

Output

All of the preceding discussion would be meaningless if there was no way to get the information back out of the computer in some useful form. The T/S 2068 has two output devices. By far the most useful is the *video display*. This is the picture that you see on the television set or monitor. Whether you use the TV or the MONITOR jack on the computer, what you see is a display that is generated by the computer. The sole purpose for this display is to present information from the computer in a manner recognizable by humans. The second form of output from the computer is through the built-in speaker underneath the T/S 2068. This device allows the computer to signal *aurally*, generating anything from a simple beep for attention, to a fully orchestrated tune. If you have a printer such as the Timex 2040 connected to the computer, then you have a third form of output—one on which the computer can generate a more permanent record called "hard copy."

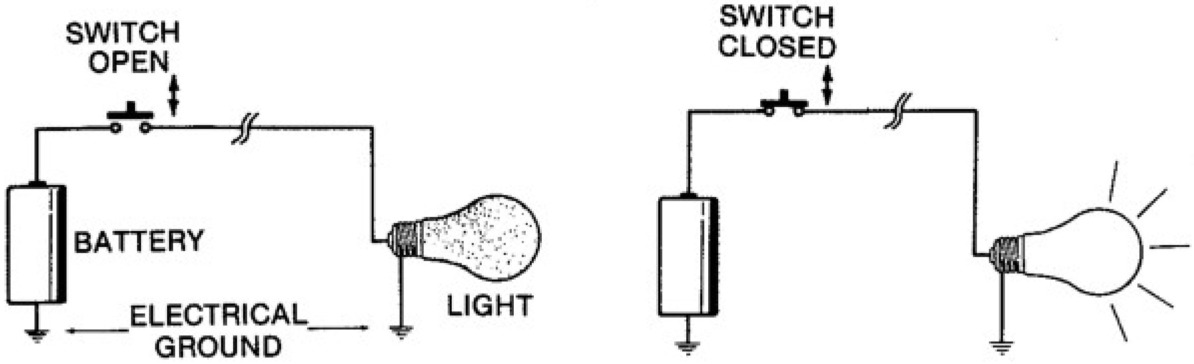
BITS OF INFORMATION

Being a digital electronic device, a computer can only deal with information as one of two forms: either the presence of a voltage (or current) or its absence. At any point in time, a single wire inside the computer can only be in one of two states. It can be connected to a source of electricity placing it at a high *potential* or it can be at a low potential otherwise known as *ground*. These two states are also referred to as +5 and ground, one and zero, or hi and low. Any point in a digital electronic circuit will be in one of these two states. When a single wire carries information from one point to another in this scheme, it carries the smallest amount of information imaginable—a bit.

Fig. 1–2 shows how a wire can transmit the status of an electric switch to a receiving indicator. There is a battery to supply the electricity, a switch to control the flow, and a light to indicate when the circuit is complete and current is flowing. In Fig. 1–2A the light is dark because the switch is open, or turned off. In Fig. 1–2B the switch has been moved to the on position completing the electrical circuit. Now the lamp glows because of the electrical current flowing through it. (Note that we have taken the liberty of defining an electrical ground to act as the return path for the current.)

A circuit such as that in Fig. 1–2, can transmit only one piece of information: either the light is on or it is off. Nonetheless, circuits similar to this are used in a variety of signaling applications—a hospital room call button, for example. By activating a switch that turns on a light at the nurse's station, you communicate the information that you need assistance.

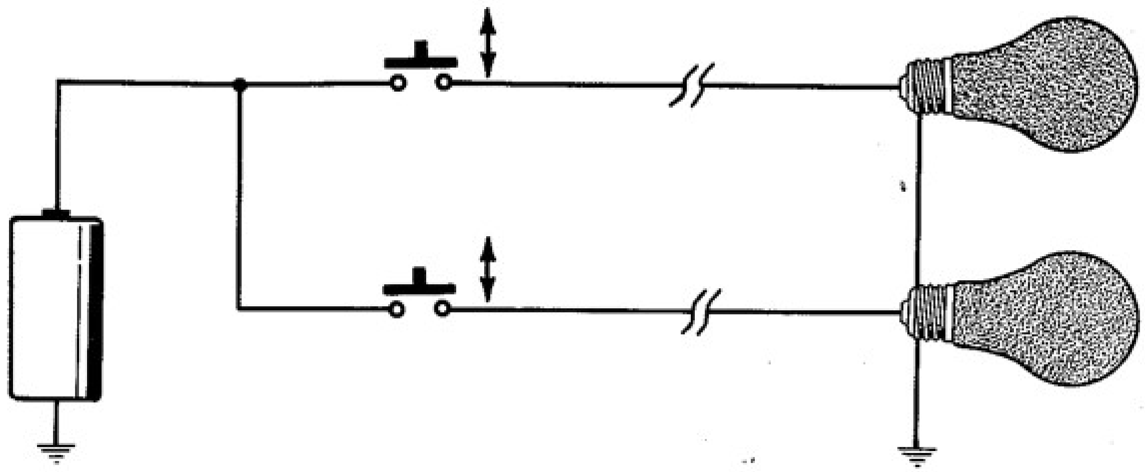
Fig. 1–3 shows how the light switch can be expanded. By adding one more wire and switch, as shown in Fig. 1–3A, it is possible to signal the four different conditions shown in Fig. 1–3B. Similarly, with three wires eight distinct patterns can be transmitted. Since each new light



A. Signal off.

B. Signal on.

Fig. 1–2. A circuit that transmits one "bit" of information.



A. Circuit.

STATE	LAMP 1	LAMP 2
1	●	☼
2	●	●
3	☼	●
4	☼	☼

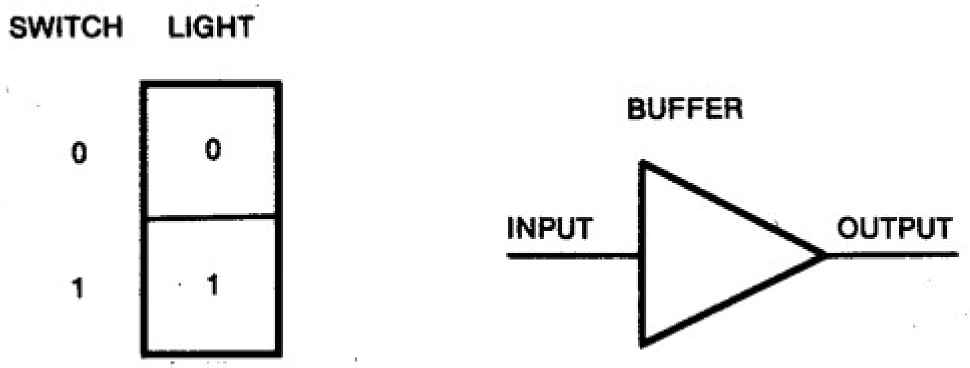
B. State table.

Fig. 1–3. Transmitting multiple bits of information.

added can be in one of two different states, it doubles the number of possible patterns. In general then, with "n" number of wires we can transmit two to the "n"th power different states. This is the basis of a *binary* system—the number system we will cover in detail in the next chapter.

BOOLEAN LOGIC

Before abandoning the light switch circuits, we'll use them to explain the basic *logic operations* performed by computers. In Fig. 1–2, we can represent pushing the switch as a logical 1 and likewise for having the lamp lit. Therefore not pushing the switch and the lamp being off are assigned the value 0. The function performed by Fig. 1–2 can then be represented by listing all of the possible switch conditions and their effect on the lamp. In this simple case, the information can be presented as shown in the state table of Fig. 1–4A. This type circuit is sometimes called a *buffer* and its logic symbol is shown in Fig. 1–4B. The *state table* is sometimes called a *transition table* and is similar to a *truth table*. Each is used to follow circuit logic.



SWITCH	LIGHT
0	0
1	1

A. Truth table.

B. Logic symbol.

Fig. 1–4. A buffer.

If the switch is changed slightly so that it is normally closed and then opens when pressed, the circuit then looks like Fig. 1–5. Keeping the same nomenclature, namely that pressing the switch is considered a logic 1, we then have a circuit known as an inverter. The truth table and logic symbol for the inverter are shown in Fig. 1–6. Note the addition of a small circle to the logic symbol which indicates that the logic state gets reversed, or inverted, when passing through this device. Such a

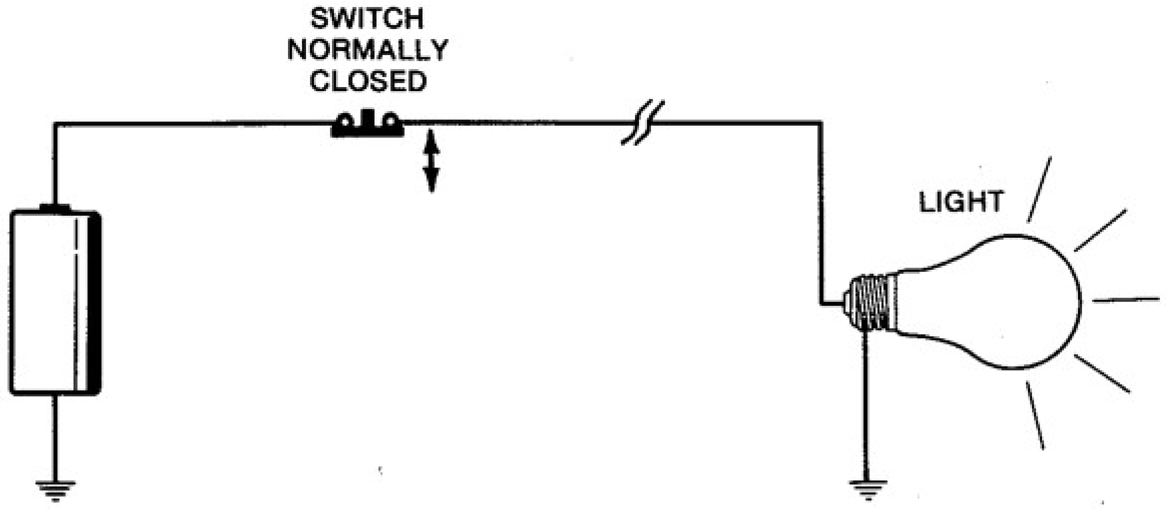
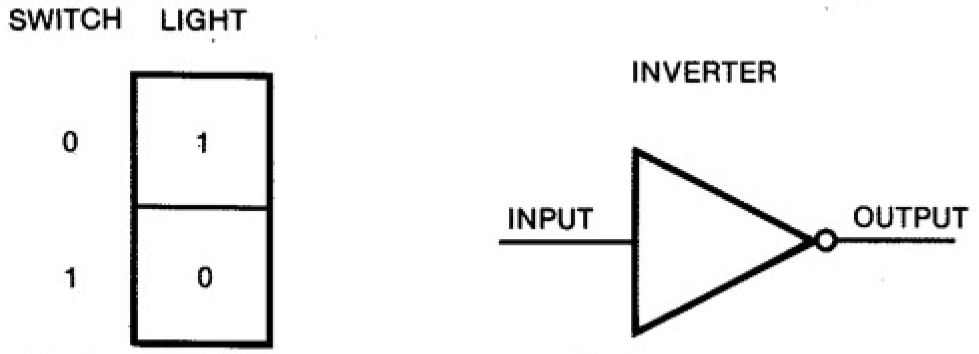


Fig. 1–5. A circuit with a different type of switch — an inverter circuit.



SWITCH	LIGHT
0	1
1	0

A. Truth table.

B. Logic symbol.

Fig. 1–6. An inverter.

gate performs the logical NOT function—that is, it puts out the opposite value from what is fed in.

The two types of logic elements discussed so far only have a single input and output. More useful functions can be derived using more than one input. In our analogy, this means adding more switches but keeping only one light. For example, another switch can be connected as shown in Fig. 1–7. In electronics lingo these switches are said to be in parallel. The result of wiring two switches in this manner is to allow pressing either switch to cause the lamp to light. It does not matter whether switch A or switch B is pressed. (The lamp will also light if both are pressed.) Since either one switch or the other can light the lamp, this circuit performs the logical OR function and therefore is called an OR gate. The truth table and symbol are shown in Fig. 1–8. To read this table, locate the row that corresponds to the states of switch A. Then find the column that corresponds to the state of switch B. At the intersection of this row and column there is a logic value which represents the output of the gate.

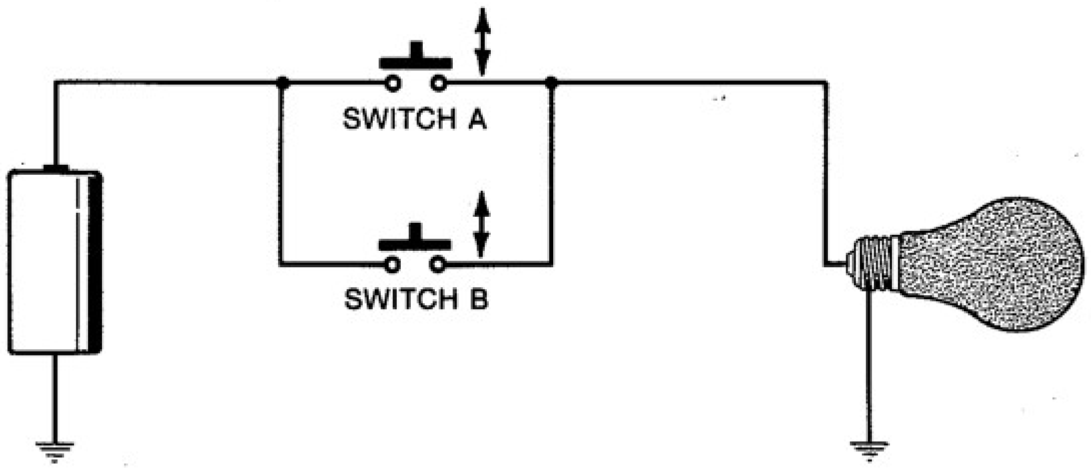
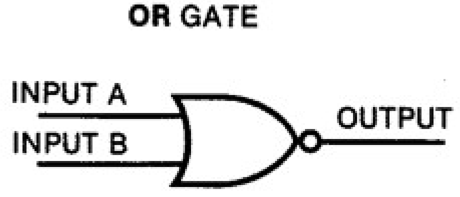


Fig. 1–7. A circuit with two switches in parallel — an OR circuit.

		SWITCH B	
		0	1
SWITCH A	0	0	1
	1	1	1

A. Truth table.



B. Logic symbol.

Fig. 1–8. An OR gate.

Another way of adding a second switch is shown in Fig. 1–9. This places the two switches in series and, therefore, requires that both switches be pressed in for the lamp to light. In this case, the circuit performs the logical AND function, since both input A *and* input B must be in the logic 1 state for the output to be a 1. The truth table and logic symbol for the AND gate are shown in Fig. 1–10.

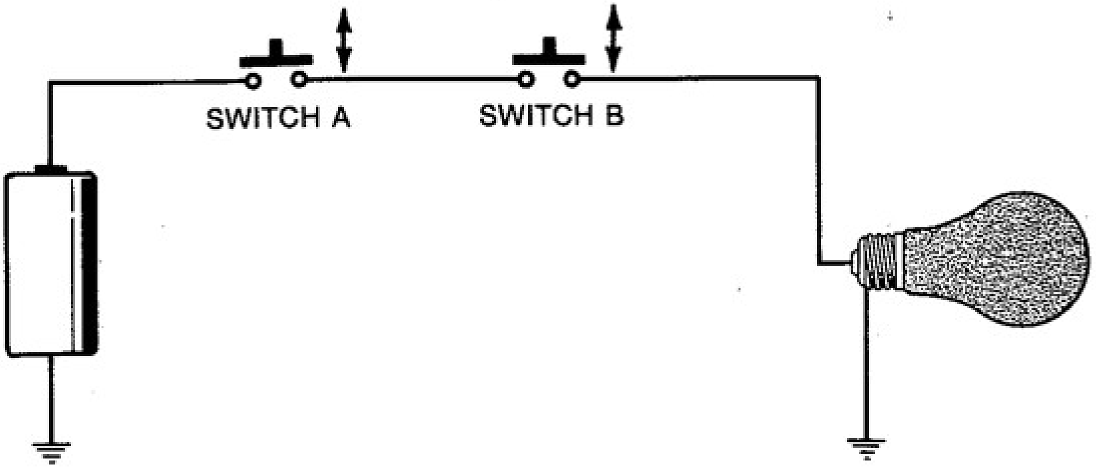
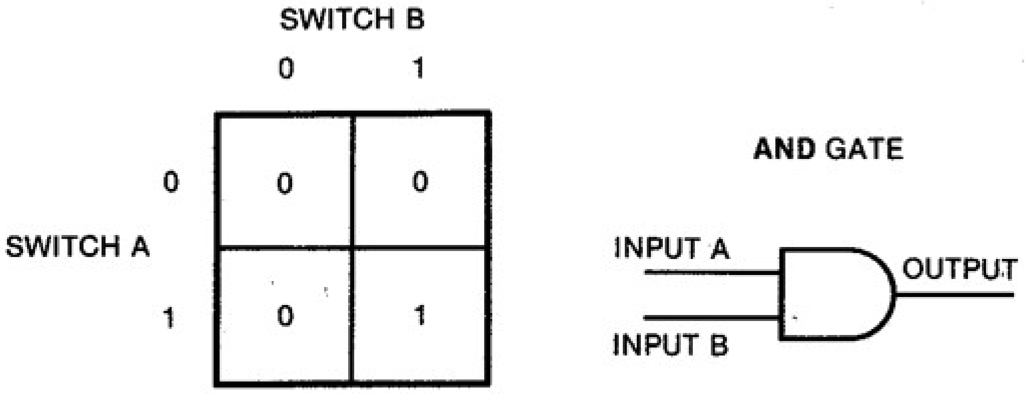


Fig. 1–9. A circuit with two switches in series — an AND circuit.



		SWITCH B	
		0	1
SWITCH A	0	0	0
	1	0	1

A. Truth table.

B. Logic symbol.

Fig. 1–10. An AND gate.

Note that it is also possible to have more than two inputs for one gate as shown in Fig. 1–11. In this case, all four inputs must "go high" for the output to be high. Often it'is convenient to add an inverter to the OR and AND gates creating the NOR and NAND gates as shown in Fig. 1–12.

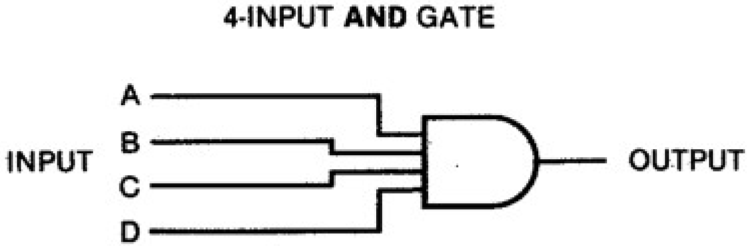
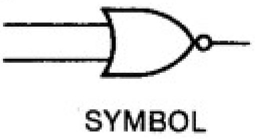


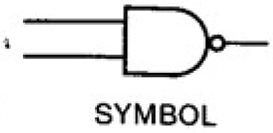
Fig. 1–11. A four-input AND gate.

	0	1
0	1	0
1	0	0



A. NOR gate.

	0	1
0	1	1
1	1	0



B. NAND gate.

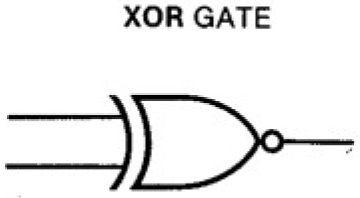
Fig. 1–12. Truth tables and logic symbols for OR and AND gates with inverters.

The three logic elements — AND, OR, and NOT — are the building blocks of all digital circuits. It can be proven that all other logical operations can be built from these three simple blocks. For example, an Exclusive OR (or XOR) device must produce a high output when either one of its inputs, *but not both,* goes high.

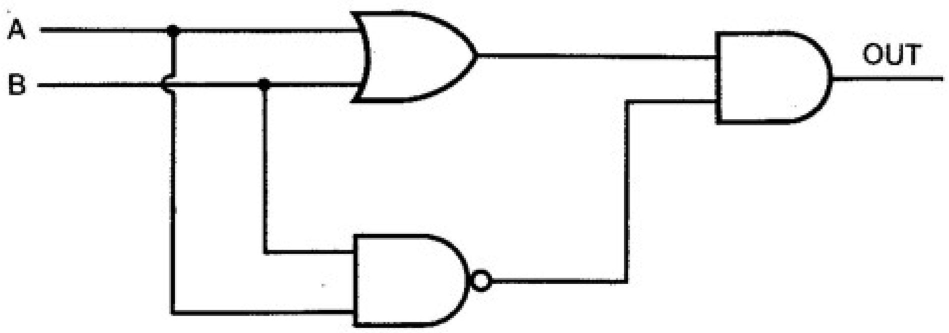
The three logic elements — AND, OR, and NOT — are the building blocks of all digital circuits. It can be proven that all other logical operations can be built from these three simple blocks. For example, an Exclusive OR (or XOR) device must produce a high output when either one of its inputs, *but not both*, goes high. The truth table and symbol for this device are shown in Fig. 1–13 along with a diagram showing how this circuit can be derived from the primitive gates.

	0	1
0	0	1
1	1	0

A. Truth table.



B. Logic symbol.



C. XOR gate using OR, AND, and NAND gates.

Fig. 1–13. The Exclusive OR (XOR) gate.

2
The Binary Number System

INTRODUCTION

Since most computers (including the T/S 2068) work with two-state electronic circuits, any information to be processed must first be converted into a series of binary numbers. Therefore it is important to understand fully the base two-number system. As shown in the preceding chapter, the brains of the T/S 2068 reside in the Central Processing Unit (CPU). All of the operations performed by the CPU are based upon the binary system.

The purpose of any number system is to convey information about quantity, just as languages are used to convey thoughts and ideas. Although many different languages exist, there is no one "correct" language. When you say "hello" to someone, you expect him to understand it as a greeting. When visiting France, however, your greeting might be met with strange looks unless you said "bonjour."

In view of the diverse languages in existence today, it is quite remarkable that one number system is used almost universally. This is the base 10 (decimal) number system that most people start learning before they can even read. This is not the only possible system, and certainly not the most convenient in many cases. Another number system you may be familiar with is the Roman Numeral system. This system undoubtedly lost its appeal because it does not have a precise mathematical foundation. (Have you ever tried to multiply XVII times IV?) Since mathematics is so closely tied with numbers, a concise and logical numbering system is essential.

Probably because we have ten fingers (which are easy to count on), our number system is based upon the number 10. We learn to count, and thus express quantity, using 10 digits. These digits are written using the numerals 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. When expressing numbers greater than nine, we need to add another digit, or place, to the left of the first number. Thus after 9 we write 10 which means one group of ten plus zero more. Then comes 11 (one group of ten plus one more), 12, 13 . . ., etc. This continues up to 99 (nine groups of ten plus nine more) after which we add another digit. The number 349 really means three hundred plus forty plus nine, as shown in Fig. 2–1.

Notice that when numbers are written this way, the first digit (the one on the left) carries the most importance or significance. The digit on the right is the least significant. That is, if the 9 became an 8, there would only be one less; if the 3 were changed to a 2, however, it would mean a loss of one hundred — certainly more significant. It should also be noted that the value of each place in our numbering system is simply a power of 10. That is, the last number represents how many ten to the zero power* or ones there are in the number. The next number represents how many ten to the one power, or tens, there are. Then comes the ten to the two power (10 × 10), or hundreds, etc. This process is continued until there are enough places to express any number desired.

With this understanding, it is then very easy to describe other number systems. Using the same procedure, we can simply change

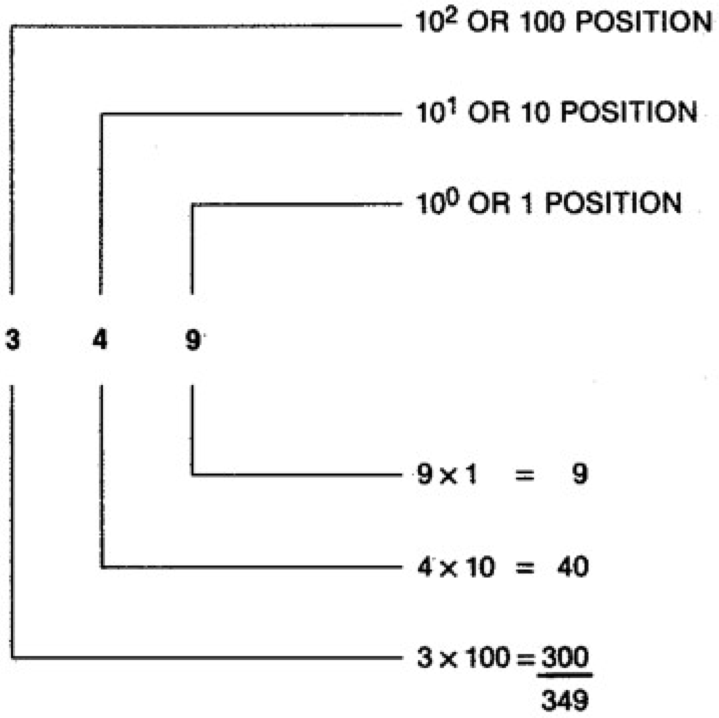


Fig. 2–1. Decimal notation

*Any number raised to the zero power equals one.

the base number, also known as the *radix*, from 10 to any other value and come up with a new numbering system. In particular, since computers only have two states with which to count (i.e., two fingers), it would seem more natural for computers to use a base 2 number system. This is commonly known as the *binary number system*. With only two numerals, a single digit number can only count from 0 to 1.

Using the standard number system format just described, it is possible to write numbers of any given size. First we must determine the value or weight of each numeral in a binary number. Since humans find it easier to think in terms of decimal numbers, it is helpful to represent each binary column as its decimal weight. Thus, the binary number 11100101 is interpreted as shown in Fig. 2–2. For larger

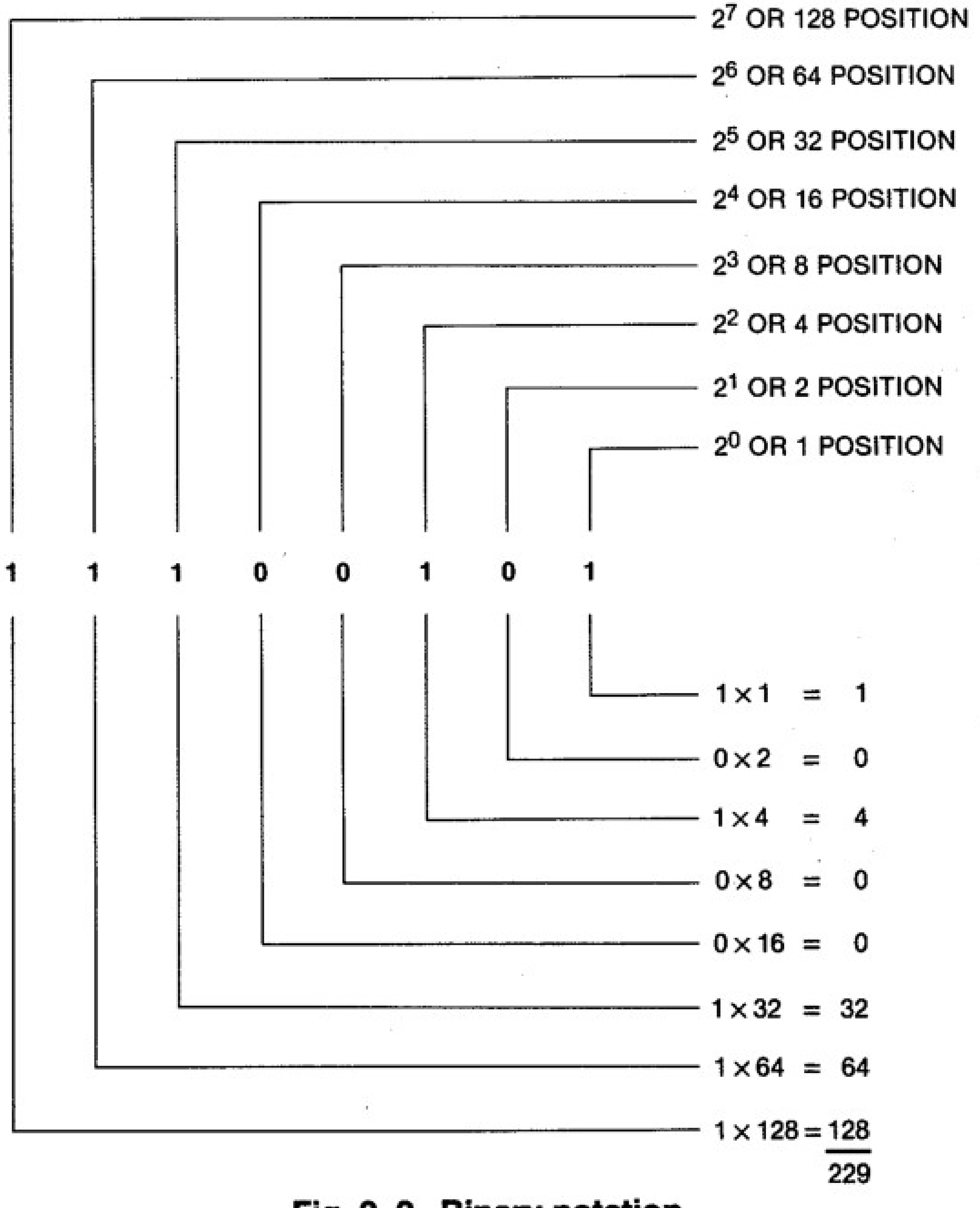


Fig. 2–2. Binary notation.

Table 2–1. Negative and Positive Power of Two

2^{-N}	N	2^N
1	0	1
0.5	1	2
0.25	2	4
0.125	3	8
0.0625	4	16
0.03125	5	32
0.015625	6	64
0.0078125	7	128
0.00390625	8	256
0.001953125	9	512
0.0009765625	10	1024
0.00048828125	11	2048
0.000244140625	12	4096
0.0001220703125	13	8192
0.00006103515625	14	16384
0.000030517578125	15	32768
0.0000152587890625	16	65536
0.00000762939453125	17	131072
0.000003814697265625	18	262144
0.0000019073486328125	19	524288
0.00000095367431640625	20	1048576

numbers, higher powers of two can be taken from Table 2–1. Note that whenever there is doubt as to the base of a given number, a small subscript will be placed after the number to indicate the radix used.

CONVERTING BINARY NUMBERS TO DECIMAL

Having defined a precise format for all number systems, including base 10 (decimal) and base 2 (binary), it should not be difficult to convert a number from one base to another. For example, we have just shown that converting a binary number to decimal involves nothing more than writing down the powers of two that correspond to each "1" digit in the binary number and then adding them all up. This is quite simple, but if you have a lot of numbers to convert, it can get quite tedious (especially when dealing with memory addresses that are 16 binary digits long). Fortunately, we have a computer — and computers are wonderful for doing tedious things. Therefore, we will write a program for the T/S 2068 that will convert binary numbers to their decimal equivalents.

A binary number must be input to start the program, and it proves easier to decipher this number if it is treated as a string. This preserves any leading zeros (which would be lost if we treated it as a numeric input). This also allows each digit to be examined using the string "slicing" capabilities of the T/S BASIC. The next step involves looking at each digit entered and keeping a running total of all binary places that contain a 1. For each 1 found, the value of 2 raised to the appropriate power is added to the total. Raising to a power is accomplished in this program using the exponentiation function denoted by the ^ symbol.

Listing 2–1 shows the simple binary to decimal conversion program. It starts by initializing a running total variable, a, in line 10. Line 20 accepts the desired binary number to be converted and assigns it to the string variable a$. Line 30 gets the number of digits entered which we will need later and line 40 checks to see if we are done (by entering no number). Line 50 sets up a loop to look at each character in the string and along with line 60 determines if that digit is zero. Note the formula (l – i) in the slicer which causes the search to proceed from right to left instead of left to right as normal. Also note the comparison to the string constant 0 instead of the number; this is due to the use of a string variable for the input number.

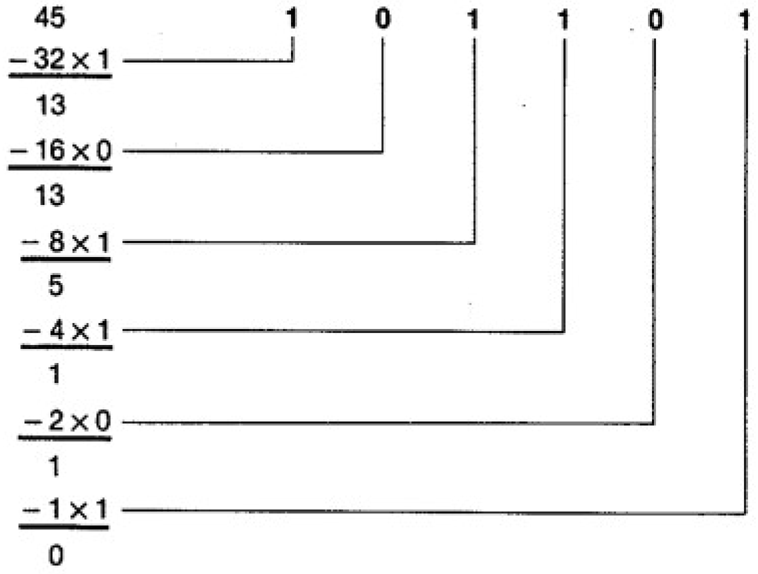
If the current digit is not a zero, then line 70 executes, adding the designated power of two to the total. Otherwise, the program skips to line 100 where the next digit is inspected. After all of the digits have been checked, the program finishes by printing out both the binary and decimal formats for the number.

Listing 2–1

```
 1Ø   LET a=Ø
 2Ø   INPUT a$
 3Ø   LET l=LEN a$
 4Ø   IF l=Ø THEN STOP
 5Ø   FOR i=Ø TO l-1
 6Ø   IF a$(l-i)="Ø" THEN GO TO 1ØØ
 7Ø   LET a=a+2↑i
1ØØ   NEXT i
15Ø   PRINT a$,a
2ØØ   GO TO 1Ø
```

CONVERTING DECIMAL NUMBERS TO BINARY

Converting decimal numbers to binary is only slightly more complicated. Whereas the previous conversion proceeded from right to left and involved addition, we now use the reverse of this technique. Therefore, the first step is to find the largest power of two that is still smaller than the given number. Actually, we could start with any large power of two, but that would only give us many "leading zeros" in front of the answer and these are usually meaningless. Having found the proper factor as described above, we can then write down a 1 as the first digit of the answer. Next, subtract this amount from the original number leaving some positive remainder. (This remainder must be less than the previous factor or we did not start with the correct power of two.) We then continue with each of the smaller binary factors. If the remainder is less than the current factor, write down a 0 for that digit. Otherwise write down a 1 and subtract that factor from the remaining value. This process continues until the last binary digit, or *bit*, is reached. For example, Fig. 2–3 shows how the decimal number 45 would be converted into binary. Of course, after the conversion is done, we can add any number of leading zeros to normalize the result for eight or sixteen places.



$45_{10} = 101101_2$

Fig. 2–3. Converting a decimal number to binary by subtraction.

Once again, we can leave the tedium to our computer. Although the conversion from decimal to binary seems more difficult for us, notice that the program in Listing 2–2 is actually more compact.

Listing 2–2

```
1Ø   INPUT a: PRINT a,
2Ø   FOR i= 15 TO Ø STEP -1
3Ø   IF a<2↑i THEN GO TO 7Ø
4Ø   PRINT "1";
5Ø   LET a=a-2↑i
6Ø   GO TO 8Ø
7Ø   PRINT "Ø";
8Ø   NEXT i
9Ø   GO TO 1Ø
```

BINARY MATHEMATICS

Having covered the complete foundation for the binary number system, we shall now investigate the subject of binary arithmetic. In many ways, binary calculations are easier to perform due to the limited size for each digit.

Addition

Probably the easiest way to describe binary addition is by summarizing the results of all possible two-number addition problems:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0 \text{ plus carry} \end{aligned}$$

You may see a similarity between these results and the discussion of the Exclusive-OR function from the last chapter. You should begin to see now how a bunch of wires and switches can actually perform something useful like adding numbers. With this set of rules, it is possible to add any two binary numbers of arbitrary length. Simply start at the right column and add each pair of numbers just as we add decimal numbers. Likewise, whenever there is a carry into the next digit it gets added with the other digits in that column. Of course, now there are three numbers to add; but this proves to be no more difficult. If one of the numbers is zero, then the result can be found by adding the other two numbers. When adding three 1's, the answer is simply 1 plus a carry into the next place. A few examples should make this clearer:

```
        |               ||||          |||||||      ◀ carries
     11ØØ1Ø1Ø        ØØ1Ø11Ø1          11Ø1111Ø
   + ØØ1Ø11ØØ      + 11ØØØ111        + Ø111Ø11Ø
   ----------      ----------        ----------
     1111Ø11Ø        1111Ø1ØØ         1Ø1Ø1Ø1ØØ
```

Subtraction

If we were to extend this line of thinking, we would next need a set of rules to define subtraction. Instead of having a "carry" bit to worry about, we would now have to deal with the possibility of a "borrow" condition. For example, when subtracting 297 from 481, the problem is written like this:

```
   481   ◀ minuend
 – 297   ◀ subtrahend
 -----
   184   ◀ remainder
```

We start by trying to subtract 7 from 1. Since that would leave us with a negative number, we *borrow* ten from next column, instead, changing the 8 to a 7. Then we subtract the first 7 from 10 + 1, or 11, resulting in the answer of 4. We then repeat this process until all columns have been subtracted. Hopefully, we do not run out of columns to borrow from (if this happens, then the subtrahend was actually larger than the minuend and we should perform the opposite subtraction and give the result a minus sign).

It turns out that there is a much easier way to perform subtraction of binary numbers (at least as far as computers are concerned). By converting the subtrahend to a different format — called its complement — we can transform any subtraction problem into an addition problem. This proves to be quite efficient and it also solves another problem — that of writing *negative* numbers.

Negative Numbers

There often comes a time when we must express a number less than zero. We have learned to write such negative numbers by placing a small minus sign in front of the first digit. Then when we add negative numbers to positive numbers, we actually perform subtraction on the negative ones. While this seems to work conveniently for humans, the computer has no comprehension of a minus sign.

One way to represent negative numbers would be to use one bit of the number to represent the sign — say the left-most bit being zero for positive numbers and a one for negative numbers. Using such a scheme, the number 45 would be written as 00101101, while negative 45 would be written as 10101101. Note that the first bit indicates the sign, leaving only seven bits remaining for the number. Thus an eight bit byte can represent a signed decimal number between −127 and +127. The problem with this approach is that adding a negative and positive number of the same magnitude does not give zero as expected. That is,

$$\begin{array}{rl} 00101101 & (+45_{10}) \\ +\ \ 10101101 & (-45_{10}) \\ \hline 11011010 & (-90_{10}) \blacktriangleleft \text{ wrong!!} \end{array}$$

There is another way of denoting negative numbers which solves this problem. (While it may seem more complicated to us, it simplifies things for the computer.) The first step in creating a negative number is to complement each bit of the positive binary number. That is, change each 0 to a 1 and every 1 to a 0. For example, the number 45 becomes:

$$\begin{array}{ll} 00101101 & (45_{10}) \\ 11010010 & (\text{each bit complemented}) \end{array}$$

The next step is to add one to the number giving us:

$$\begin{array}{rl} 11010010 & \\ +\qquad\quad 1 & \\ \hline 11010011 & (-45_{10}) \end{array}$$

This result is called the twos complement of the original number. We will use it to represent negative numbers. Just to be sure that you understand, let's calculate the twos complement of a couple more numbers:

$$\begin{array}{rcccr}
\text{Ø11ØØ1Ø1} & (101_{10}) & & (217_{10}) & \text{11Ø11ØØ1} \\
\text{1ØØ11Ø1Ø} & \blacktriangleleft & \text{complement} & \blacktriangleright & \text{ØØ1ØØ11Ø} \\
+ \qquad\qquad 1 & \blacktriangleleft & \text{add one} & \blacktriangleright & + \qquad\qquad 1 \\
\hline
\text{1ØØ11Ø11} & (-101_{10}) & & (-217_{10}) & \text{ØØ1ØØ111}
\end{array}$$

The twos complement of a number has great significance for binary numbers. For one thing, it has the correct property of negative numbers in that adding any negative number to its positive counterpart yields a result of zero:

$$\begin{array}{rl}
\text{ØØ1Ø11Ø1} & (+45_{10}) \\
+ \ \text{11Ø1ØØ11} & (-45_{10}) \\
\hline
\text{1ØØØØØØØØ} & (\text{Ø, ignoring carry})
\end{array}$$

Notice that the result does equal zero if we ignore the carry into the next higher position. Actually it is quite easy to show that adding any number to its twos complement equals zero, according to the way in which we have derived the twos complement number. Since we started by taking the complement of the number, it should be obvious that any number plus its complement equals a binary 1 1 1 1 1 1 1 1, for example:

$$\begin{array}{rl}
\text{ØØ1Ø11Ø1} & \blacktriangleleft \ \text{any number} \\
+ \ \text{11Ø1ØØ1Ø} & \blacktriangleleft \ \text{plus its complement} \\
\hline
\text{11111111} & \blacktriangleleft \ \text{equals all ones}
\end{array}$$

Therefore adding the twos complement to a number must equal this plus one or:

$$\begin{array}{r}
\text{11111111} \\
+ \qquad\qquad 1 \\
\hline
\text{1ØØØØØØØØ}
\end{array}$$

This proves that the twos complement notation satisfies the condition that adding any number to its negative value equals zero. Furthermore we can show that adding any numbers, regardless of their signs, may be accomplished with this scheme. For example:

$$\begin{array}{rr}
\text{ØØ1Ø11Ø1} & (+45_{10}) \\
+\ \text{1111Ø11Ø} & (-10_{10}) \\
\hline
\text{ØØ1ØØØ11} & (+35_{10})
\end{array}$$

Notice again that when dealing with negative numbers we will ignore any carry out to the ninth binary place. Since twos complement notation correctly follows the algebraic rules for negative numbers, it becomes the natural choice. It also makes a separate subtraction operation unnecessary. To subtract two numbers, simply form the twos complement of the subtrahend and then add.

Binary Fractions

What happens when we need to express numbers less than one but greater than zero? In modern thinking, these are known as fractions. You probably know two different ways of expressing fractions. One way is to divide the whole number "1" into several equal parts and then express how many of these portions there are. For example, we might say add "one-third" of a cup of sugar. When the partial units chosen become a negative power of the base 10 system, we can then express the number as a decimal fraction. For example, one-tenth is commonly written as 0.1 where the decimal point signifies that the numbers to the right are fractions.

We extend the decimal system by adding places for the negative powers* of ten as shown in Fig. 2–4. The same thing applies to the binary number system as shown in Fig. 2–5. Table 2–1 lists some of the negative powers of two.

Multiplication

As previously mentioned, the Z80 CPU does not know anything about multiplication or division. The only mathematical functions that it can perform directly are addition and subtraction. However, it is quite easy to show the relationship between these four operations.

*Raising a number to a negative power is equivalent to dividing by the number raised to the positive power (e.g., $10^{-3} = 1/10^3$).

Multiplying any number *A* by another number *B*, is equivalent to adding A to itself B times or adding B to itself A times. That is:

$$A \times B = \overbrace{A+A+\ \ldots\ +A}^{\text{B times}} = \overbrace{B+B+\ \ldots\ +B}^{\text{A times}}$$

$$6 \times 3 = 6+6+6 = 3+3+3+3+3+3 = 18$$

So far so good, as long as we are dealing with small (one digit) numbers. But when we try to multiply 46 × 1023, this simple addition approach becomes unwieldy:

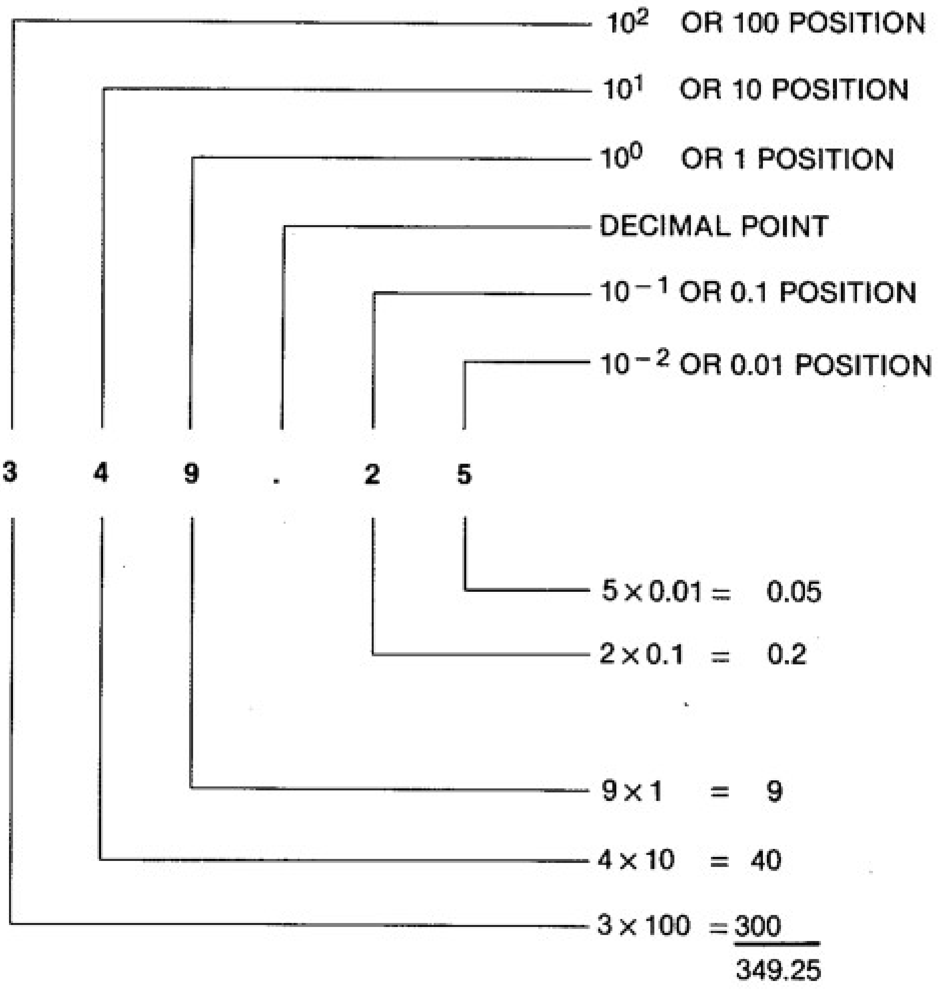


Fig. 2–4. Extending decimal notation to include fractions.

$$46 \times 1023 = \overbrace{46+46+46+46+\ \ldots\ +46}^{\text{1023 times — whew!}}$$

Even if we could add two numbers per second, it would take quite a while to perform this simple problem. As you already know however, there is an easier way to multiply multidigit numbers. This process relies on our definition of such numbers:

$$46 = 4 \times 10 + 6$$

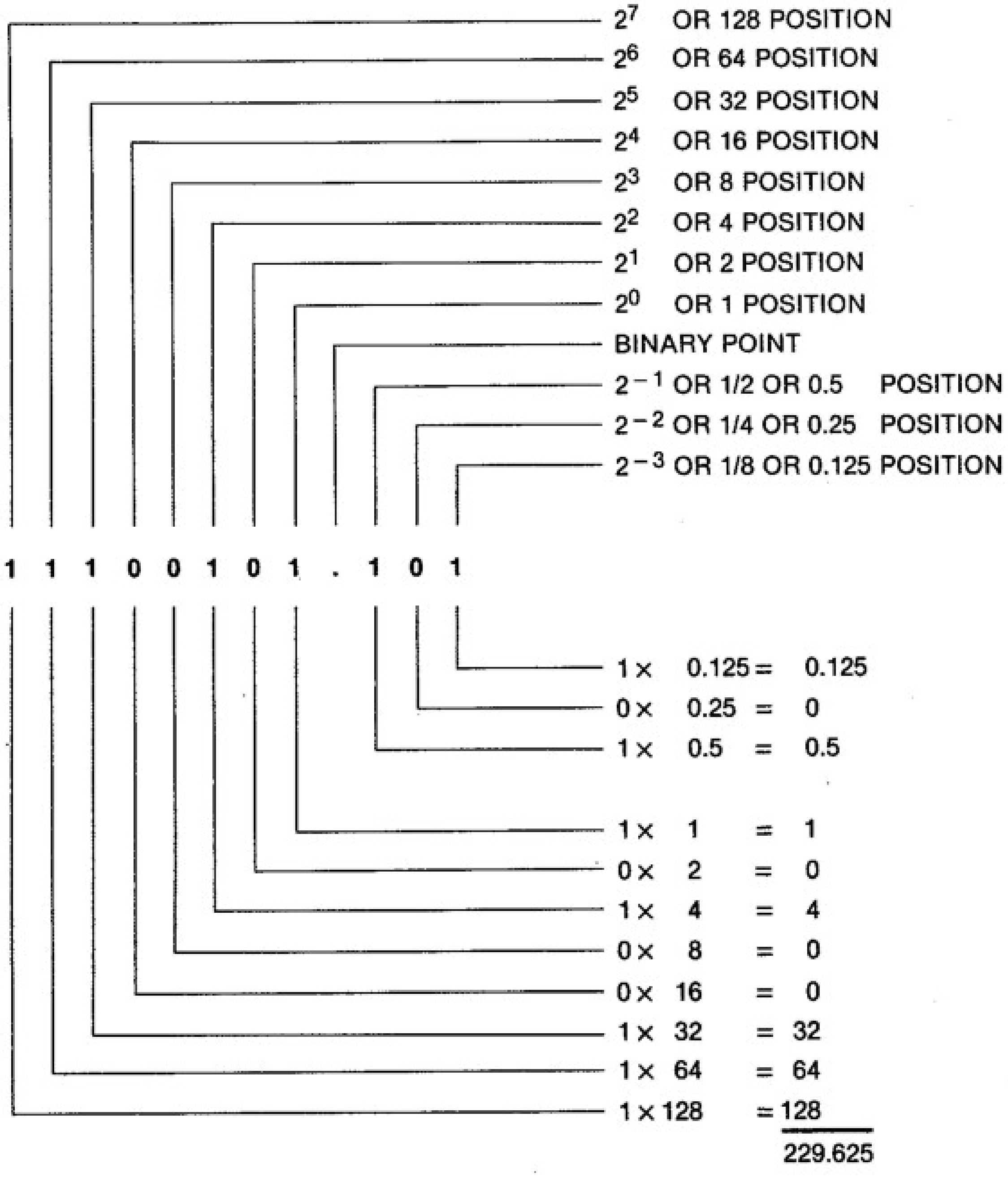


Fig. 2–5. Extending binary notation to include fractions.

Therefore:

$$1023 \times 46 = (1023 \times 4) \times 10 + 1023 \times 6$$

That is, we break down one of the multidigit factors into its component parts. This reduces the problem to a series of single-digit problems which are easier to perform. We can then add the results of these simpler problems, keeping track of their relative weights, to arrive at the final answer. Thus we would write:

$$
\begin{array}{rll}
1023 & & \leftarrow \text{multiplicand} \\
\times \quad 46 & & \leftarrow \text{multiplier} \\
\hline
6138 & \left.\begin{array}{l}(6\times1023) \\ (4\times1023)\end{array}\right\} & \begin{array}{l}\text{partial} \\ \text{products}\end{array} \\
4092 & & \\
\hline
47058 & & \leftarrow \text{product}
\end{array}
$$

Note that the second partial product is shifted over one place to the left. This gives it the proper weighting factor (× 10) necessary for the addition of the two partial products.

Binary multiplication is probably the simplest of all, as evidenced by the two-number equations next:

Ø × Ø = Ø
Ø × 1 = Ø
1 × Ø = Ø
1 × 1 = 1

Note how this corresponds to the function of an AND gate. Another way of looking at this is to say that zero times any number equals zero and that one times any number equals that number. This should sound reasonable since the same is true in the decimal system. But since the binary system stops at 1, we do not have any further multiplication "tables" to worry about. This also eliminates the problem of "carries" from one digit to another. Therefore, to multiply two binary numbers we proceed like so:

```
     1Ø111Ø1      (93_10)
  ×    1Ø11Ø      (22_10)
  ----------
     ØØØØØØØ
    1Ø111Ø1
   1Ø111Ø1     }  partial products
  ØØØØØØØ
1 Ø111Ø1
------------
1 1111111110      (2046_10)
```

Notice that every partial product will be either all zeros or the multiplicand itself. There is never any carry during the multiplication although there may be when adding up the partial products. Thus the multiplication of binary numbers can be reduced to three simple steps:

1. Checking the value of each bit in the multiplier
2. Shifting a binary number to the left
3. Adding the result

As we will show in Chapter 12, these steps are quite easy for the Z80 to perform. With the proper machine language code, any size binary numbers can be multiplied. Finally, note that whenever two numbers are multiplied (in any number system), the resulting product can have as many digits as the combined total of the numbers being multiplied. In a computer, this means that multiplying two 8-bit numbers can yield a 16-bit result. Whenever negative numbers are to be multiplied, they are first changed to their positive values. After the multiplication is performed, the sign of the result is determined by the standard rule — if both signs were the same, the result is positive, otherwise the result is negative. In this latter case, the result is then converted back into its twos complement form.

Division

Division can be defined in terms of a similar process using subtraction. We won't bother with all of the details, but simply state that the number of subtraction operations possible (before the dividend goes negative) becomes the quotient. Anything left over becomes the remainder. We can even use the standard division notation:

```
             1Ø1      ◀ quotient
        ┌───────
  1 Ø1  │ 11Ø1Ø      ◀ divisor/dividend
          101
          ───
          ØØ11
          ØØØØ
          ────
            11Ø
            1Ø1
            ───
              1      ◀ remainder
```

There are many ways to program the Z80 to perform binary division. We'll give one example in Chapter 12.

THE HEXADECIMAL NUMBER SYSTEM

Writing binary numbers can get tedious and difficult to read. A common "shorthand notation" involves grouping the binary number into four-bit chunks. Thus an eight-bit number would break down into two chunks. With four bits, there are 16 possible values for any chunk. If we try to represent them by the decimal numerals 0–9, it leaves us with 6 more values but no more numerals. We need more symbols, so we turn to the letters of the alphabet. The next value becomes A followed by B etc. up to F. This is the basis for the base 16, or hexadecimal number system. Table 2–2 shows the decimal/hexadecimal relationship. When writing hexadecimal numbers, we will often add the letter h to indicate the base rather than using a subscripted number 16. Thus 3Bh is the same as $3B_{16}$.

Converting Hexadecimal to Decimal

Converting hexadecimal to decimal is very similar to converting binary to decimal. Instead of using the binary weight factors, we now use powers of 16 as shown in Table 2–3. Thus:

$$\begin{array}{lrlrlr}
1AE = & 1 \times & 16^2 = & 1 \times & 256 = & 256 \\
 & A \times & 16^1 = & 10 \times & 16 = & 160 \\
 & E \times & 16^0 = & 14 \times & 1 = & \underline{14} \\
 & & & & & 430
\end{array}$$

Table 2–2. Hexadecimal Number System

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

Table 2–3. Powers of 16

16^N	N	16^{-N}
1	0	1
16	1	0.0625
256	2	0.00390625
4096	3	0.000244140625
65536	4	0.0000152587890625
1048576	5	0.00000095367431640625
16777216	6	0.000000059604644775390625

Converting Decimal to Hexadecimal

The same thing holds true for converting decimal numbers to hexadecimal. You can use the subtraction method as in Fig. 2–3 or you can write the conversion like a division problem with changing divisors:

$$
\begin{array}{r|l}
 & \quad 1AE_{16} \\
\hline
1 \quad 16 \quad 256 & 430 \\
 & \underline{256} \\
 & 174 \\
 & \underline{160} \\
 & 14
\end{array}
$$

Note: All numbers are in base 10 except quotient.

OTHER BINARY REPRESENTATIONS

So far, we have shown how binary representations are used for writing numbers between 0–255 or signed numbers between −127 to +127. Obviously a computer must deal with other forms of data. Three major formats are used by almost all personal computers. The simplest involves assigning each of 128 binary values to represent each letter in the alphabet including numbers and punctuation. In this way, any type of written information can be stored within the computer in binary form. As with almost all coding schemes, there is no correct code, but the most popular is the ASCII code as shown in Appendix A. The T/S 2086 uses ASCII code to store characters. It also assigns the unused codes and the remaining 128 possible byte values to represent graphic characters, special operations, and the BASIC keywords. The entire character set for the T/S 2068 is shown in Table 2–4.

Another data structure can be used to store integer numbers greater than 256. This requires more than eight bits, so we use two complete bytes. This gives us 16 bits to work with for a total of 65,536 different values. This should suffice for all but the largest numbers.

Numbers greater than 65,536 require more bytes to store them. To represent very large numbers, another number format is used. This format is called scientific notation and we will show how the computer handles such numbers in the next chapter.

Finally, when we introduce machine language programming we will come across another form of binary representation called binary coded decimal (BCD). This notation allows two decimal numbers to be

Table 2–4. T/S 2068 Character Set (ASCII Compatible)

Code	Character
Ø	Not used
1	Not used
2	Not used
3	Not used
4	Not used
5	Not used
6	PRINT comma
7	EDIT
8	Cursor Left
9	Cursor Right
1Ø	Cursor Down
11	Cursor Up
12	DELETE
13	ENTER
14	Number (slug)
15	Not used
16	INK control
17	PAPER Control
18	FLASH Control
19	BRIGHT Control
2Ø	INVERSE Control
21	OVER Control
22	AT Control
23	TAB Control
24	Not used
25	Not used
26	Not used
27	Not used
28	Not used
29	Not used
3Ø	Not used
31	Not used

Code	Character
32	Space
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	–
46	.
47	/
48	Ø
49	1
5Ø	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
6Ø	<
61	=
62	>
63	?

Table 2–4 — cont. T/S 2068 Character Set (ASCII Compatible)

Code	Character
64	@
65	A
66	B
67	C
68	D
69	E
7Ø	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
8Ø	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
9Ø	Z
91	[
92	/
93	]
94	⬆
95	_

Code	Character
96	£
97	a
98	b
99	c
1ØØ	d
1Ø1	e
1Ø2	f
1Ø3	g
1Ø4	h
1Ø5	i
1Ø6	j
1Ø7	k
1Ø8	l
1Ø9	m
11Ø	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
12Ø	x
121	y
122	z
123	{ (ONERR)
124	STICK
125	} (SOUND)
126	FREE
127	© (RESET)

Table 2–4 —cont. T/S 2068 Character Set (ASCII Compatible)

Code	Character
128	□
129	▝
13Ø	▘
131	▀
132	▗
133	▐
134	▚
135	▜
136	▖
137	▞
138	▌
139	▛
14Ø	▄
141	▟
142	▙
143	█
144	(a)
145	(b)
146	(c)
147	(d)
148	(e)
149	(f)
15Ø	(g)
151	(h)
152	(i)
153	(j) user
154	(k) graphics
155	(l)
156	(m)
157	(n)
158	(o)
159	(p)

Code	Character
16Ø	(q)
161	(r)
162	(s)
163	(t)
164	(u)
165	RND
166	INKEY$
167	PI
168	FN
169	POINT
17Ø	SCREEN$
171	ATTR
172	AT
173	TAB
174	VAL$
175	CODE
176	VAL
177	LEN
178	SIN
179	COS
180	TAN
181	ASN
182	ACS
183	ATN
184	LN
185	EXP
186	INT
187	SQR
188	SGN
189	ABS
19Ø	PEEK
191	IN

Table 2-4 — cont. T/S 2068 Character Set (ASCII Compatible)

Code	Character
192	USR
193	STR$
194	CHR$
195	NOT
196	BIN
197	OR
198	AND
199	<=
200	>=
201	<>
202	LINE
203	THEN
204	TO
205	STEP
206	DEF FN
207	CAT
208	FORMAT
209	MOVE
210	ERASE
211	OPEN #
212	CLOSE #
213	MERGE
214	VERIFY
215	BEEP
216	CIRCLE
217	INK
218	PAPER
219	FLASH
220	BRIGHT
221	INVERSE
222	OVER
223	OUT

Code	Character
224	LPRINT
225	LLIST
226	STOP
227	READ
228	DATA
229	RESTORE
230	NEW
231	BORDER
232	CONTINUE
233	DIM
234	REM
235	FOR
236	GOTO
237	GO SUB
238	INPUT
239	LOAD
240	LIST
241	LET
242	PAUSE
243	NEXT
244	POKE
245	PRINT
246	PLOT
247	RUN
248	SAVE
249	RANDOMIZE
250	IF
251	CLS
252	DRAW
253	CLEAR
254	RETURN
255	COPY

"packed" into an 8-bit byte as shown in Fig. 2–6. While this is not as efficient as true binary notation, it can eliminate errors that creep in during the binary/decimal conversions. (With binary fractions for instance, we often have to round off to a fixed number of binary places.)

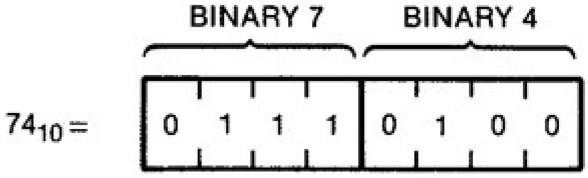


Fig. 2–6. Binary Coded Decimal notation.

3
Exploring the T/S 2068 BASIC

T/S 2068 OPERATING SYSTEM

When the T/S 2068 is first turned on, a few horizontal bars appear on the screen, possibly some flashes, and then the title page comes up with the Sinclair and Timex copyright notices. You may have assumed that the computer was just warming up, like the picture tube does in your tv set. Actually, there are no parts inside the T/S 2068 that need to warm up to operate. The instant that you turn on the power, the computer begins executing a program called the operating system that resides inside the computer in ROM.

With few exceptions, whenever power is applied to the Z80, it attempts to execute a program. In a sense, the CPU is like a robot blindly carrying out its duty. It looks at an internal counter to see what memory address holds its next instruction, reads that instruction, and then tries to perform whatever function is called for. As we will see in Chapter 4, the Z80 CPU can be forced to begin executing its program at the start of memory, namely address 0000. By placing ROM at this address, the computer can be made to perform in a predictable manner. Every time the computer is turned on it will begin executing whatever program is stored in the ROM.

Initialization

In the T/S 2068, ROM contains a program that initializes the computer and sets it up to accept commands typed on the keyboard by the user. Even after the copyright page has been displayed and the computer just seems to be sitting there, it is really still executing a program.

At the same time, another circuit inside the T/S 2068 generates the video display that you see on your tv or monitor. This circuit uses a portion of the T/S 2068's RAM to store a digital pattern similar to what is displayed. When the computer is first turned on, the contents of this portion of RAM (as well as the rest of RAM) is unknown. Almost any value can be present in any given RAM location. Because the RAM chips are very symmetrical in design, however, it is likely that many locations have the same value. Because of this, we usually see a regular pattern on the screen, such as a few horizontal bars.

As the initialization routine stored in ROM begins to execute, one of the first things it does is to check how much RAM is in the machine. In doing this, almost every byte of RAM is set to zero. You can easily check this for yourself. Turn on the computer and immediately PEEK at any location between 26800 and 65300. You will find a value of zero at all of these locations.

Initialization of the T/S 2068 also includes checking for ROM contained in a Timex Command Cartridge. Depending on the amount of RAM you have in your computer, the type of cartridge (if any), and certain other peripherals, the initialization program partitions off sections of RAM. Certain areas of RAM are reserved for the video display information, system housekeeping chores, the BASIC program and its variables, peripherals such as a printer, etc. If a cartridge has been installed, then the computer may continue by executing the program on that cartridge. Otherwise, the standard *operating system* program contained within the T/S 2068 will continue executing.

By Your Command . . .

After a few seconds, the initialization routine is completed leaving a blank screen with two copyright notices. At this point, the computer does not know what you want it to do, so it enters a routine that waits for further instructions. The only way of communicating our instructions to the computer is by typing them in on the keyboard. Therefore the major function of this routine is to determine when we have depressed a key and which key that is. Furthermore, this routine must keep track of all the keys typed until the ENTER key is pressed. This is because the computer cannot determine what you want it to do until the entire command has been entered and the ENTER key pressed.

While the computer is waiting for you to type in a command, it takes care of a few *housekeeping* chores. These include updating the video display and keeping a running "frame counter" (more on these later). Also, whenever a key is pressed and held down for more than half a second, this routine automatically simulates the action of repeatedly pressing and releasing that key. This is known as the *auto-repeat* function. The actual *delay* before a key begins to repeat and the repeat *rate* are both controlled in software. In Chapter 15, we will show how these values can be changed with a simple POKE statement.

As we continue typing in our command, the computer stores each character in a small section of RAM called the *edit buffer*. The actual location of this buffer changes as new commands or program lines are entered but the computer always keeps track of where it is. When the ENTER key is finally pressed, the operating system then begins to process the command.

The first thing that it does is to invoke a routine called a *parser* which examines the line of characters that were typed and which are now sitting in the edit buffer. It is the parser's job to figure out what command, if any, is being given to the computer. The first task of the parser is to determine whether a direct command or a BASIC program line has been entered. This is done by looking at the first character in the line. If it is a number, then the entire line is interpreted as a BASIC program line; otherwise, it must be a direct command. We'll discuss the latter case first.

After determining that this is a direct command, the parser continues to scan each character of the line. If the entry contains any syntax errors, it is reported by placing a question mark in front of the offending character. Assuming there are no errors, the parser then determines what action to take and transfers control to the proper routine in the system ROM. Each of the T/S 2068 commands has an appropriate machine language program, or *subroutine*, which handles that command. If the command requires further data such as a variable name, constant, numeric expression, etc., then the parser continues scanning the buffer for these pieces.

A TUTORIAL ON THE T/S 2068 BASIC INTERPRETER

Most of the system ROM contains a program called the BASIC interpreter. This is the program that allows us to communicate with the T/S 2068 in an English-like manner. The interpreter program is written in machine language and integrated into the operating system. An interpreter program is interesting in that it allows us to write and execute programs using a higher level language such as BASIC. But as far as the CPU is concerned, it is always running the interpreter program. The job of the interpreter, therefore, is to translate BASIC commands into an appropriate sequence of machine language instructions.

One of the advantages of using a higher level language is that you do not have to keep track of absolute memory locations. When writing a program and/or storing data in the computer, everything gets broken down into a series of bytes. Since all 65,535 bytes look alike to the Z80, there is a problem in keeping track of which bytes hold what information. When you write programs in machine language, it is up to you to keep things straight. This can become very complicated and it is a major reason why higher level languages were developed. In BASIC for instance, we only have to say:

```
LET a = 3
```

to create a variable in RAM and store the value of three there. We have no idea as to the exact memory location(s) where "a" is to be found, but we can always get the current value of the variable by simply referring to it as "a." This, of course, makes programming much easier but limits the program's speed and versatility. One way to get around these limitations is to include machine language routines in your BASIC programs (or, of course, write the programs entirely in machine language). Part Two of this book is devoted to this subject. First, however, we'll examine the T/S 2068 BASIC interpreter in more detail.

At this point, we can begin to talk about the specifics of the T/S 2068 BASIC interpreter and how it works. Fortunately, we can use the computer itself to explain many of these functions. To do this, we will make extensive use of the PEEK and POKE statements. This is because we need to work with absolute memory locations and the T/S 2068 only speaks BASIC when it is first turned on. Just to make sure that you understand PEEK and POKE, let's take a moment to look at them in some detail.

The PEEK Command

The PEEK command is used to look at the contents of a single byte in memory. Any valid address (in the range of 0–65,535) can be examined. It does not matter what type of memory is located at that address. The Z80 CPU will simply go to that location and see what data is there. If there is no circuit device mapped into that address, then the data read by the Z80 will be meaningless. If there is RAM or ROM there, then the information (one byte's worth) will be read out by the Z80 and then relayed to the program or display.

Since all data read by the PEEK statement consists of one byte, the result will always be in the range of 0-255. This result can be printed on the screen by a command such as:

```
PRINT PEEK Ø
```

The result of entering this command is the number 243 printed on the screen. This is the value stored at that location and it will always be there. That is because this data is stored in ROM and therefore all T/S 2068s will display the same results. There are actually over 16,000 bytes to choose from for which we can predict the PEEK results.

The result from a PEEK can also be assigned to a variable, for example:

```
LET a = PEEK 1ØØØØ
```

Quite often, computers must deal with numbers greater than 255. A 16-bit memory address, for example, requires two adjoining bytes. One byte holds the lower eight bits of the address or the *least significant byte*. The next memory location then contains the upper eight bits or the *most significant byte*. These are also referred to as the *low* and *high* bytes respectively. Although we would normally write the high byte first followed by the low byte, in most cases, the computer finds it more convenient to store the low byte first. It is the location of this byte that is usually referred to when describing the location of a 2-byte number. The high byte is therefore found at the next *higher* memory location.

When using the PEEK function to examine a 16-bit number, we must use the formula:

```
PEEK n + 256 * PEEK (n+1)
```

For example, to find the starting address of a BASIC program in the T/S 2068, we would type:

```
PRINT PEEK 23635 + 256 * PEEK 23636
```

The POKE Command

The POKE command allows us to place whatever value we desire into any RAM location. Although we could POKE to any valid memory address, unless there is RAM located at that address (or some other device capable of storing information) the data POKEd will be lost. For example, if we POKE to a ROM location, nothing much will happen. Try:

```
POKE Ø,Ø
```

Then:

```
PRINT PEEK Ø
```

Note that the data at location 0 has not changed from the previous PEEK. This data is permanently stored in the computer's ROM and cannot be changed. Also note that the POKE command requires two parameters: the *address* to be changed and the *data* to be put there. Again, the address must be valid (0-65,535) as well as the data (0-255). Otherwise, an "integer out of range" error will be reported.

If we try the last example with a "good" memory address (i.e., one containing RAM and not reserved for system use), we can verify the results of a POKE command.

```
POKE 3ØØØØ,1Ø
PRINT PEEK 3ØØØØ
POKE 3ØØØØ,33
PRINT PEEK 3ØØØØ
   etc.
```

There are many ways to POKE 16-bit numbers, but none of them are elegant. One way to POKE a 16-bit value "data" into the two bytes starting at "address" would be:

```
LET temp = INT (data/256)
POKE address, data-temp*256: POKE address + 1, temp
```

Now that we have the digging tools (PEEK and POKE), we are ready to explore the inner workings of the T/S 2068 BASIC. Our exploration will be divided into two parts since there are two parts to any BASIC program: the program itself and the variables, or data, that it uses.

These two items are stored in different areas of the computer's RAM and the first job will be to find out where they are. Their location depends on a variety of factors, and indeed, as we use the computer (entering program lines for example), these areas can even *move around!* Fortunately, we know that the computer must keep track of their location. It does this by maintaining a table of *pointers,* sort of an index, to various memory locations that it needs. These pointers are stored in the RAM area reserved for system use. A complete list can be found in Appendix D of the *T/S 2068 User Manual,* but for now we will only be concerned with two of them. The first is called PROG and is located at address 23635 (plus 23636, since it is a two-byte number). This pointer always indicates the memory location where the image of your BASIC program resides. We will cover this in more detail later.

The other pointer we will need is called VARS. As you might guess, this tells us where the variables are located. By the way, names such as PROG and VARS are used only for convenience — to help you remember what they are used for. These terms have no special meaning to the computer.

VARIABLE STORAGE

CAUTION: As you follow along with your computer, it is imperative that you enter each example precisely. Do not skip any or try some different wording. If you do, the memory addresses given will not be valid.

Integers

If we type the command:

```
LET a = 1
```

into the computer we know that the BASIC interpreter will have to create a variable called *a* and give it a value of 1. *Creating* a variable is really nothing more than setting aside some memory for its value and keeping track of the variable's *name* and where it is located. An easy way to do this is by keeping all variables together in a sort of *list.* That is, as more variables are needed, simply add them to the bottom of the list by placing them in the next higher memory location. Then as long as we make it clear where one variable starts and another ends, we only have to keep track of the starting address for this list.

This is exactly what the T/S 2068 does as we will demonstrate with the following exercise. When the computer has just been turned on, there will be no variable list, since no variables have been defined yet. The beginning address for this table has been determined, however, and we can find this out by entering:

```
PRINT PEEK 23627 + 256 * 23628
```

The number 26710 should appear after you press ENTER. Thus the number 26710 is the starting address for a list of variables to be stored in memory. To place some variables into the list we can start by entering:

```
LET a = 1
```

The computer responds with:

```
Ø OK, Ø:1
```

which, of course, just means that it has finished your command and there were no errors. Although nothing else appears to have happened, we can check that a variable called 'a' has in fact been created with a value of 1.

Since we know the starting address for the variable list, it stands to reason that the variable "a" should be found there. So type:

```
PRINT PEEK 2671Ø
```

If you've done everything correctly so far the result should be 97. If we check the T/S 2068 character set in Table 2–5, we will find that 97 is indeed the code for the letter "a." So it would seem that the first byte represents the name of the variable. The value of "a" must also be stored here, so let's do a few more PEEKS:

```
PRINT PEEK 26711 (Ø)
PRINT PEEK 26712 (Ø)
PRINT PEEK 26713 (1)
PRINT PEEK 26714 (Ø)
PRINT PEEK 26715 (Ø)
PRINT PEEK 26716 (128)
```

The number in parentheses is the result you should get from entering each statement. Typing in all of those PRINT statements is a bit tedious. You may have realized that it could be done easier by using a FOR . . . NEXT loop. This is true, but it would require us to define another variable which we don't want to do quite yet. And, if you had tried this by writing a BASIC *program*, the entire variable list would move to a completely new location in memory!

Before we explain the results of these PEEK statements, it is important to realize that there are three types of variables allowed in BASIC: *whole numbers* (integers), *real numbers* (floating point), and *strings* (characters). Each of these variable types use a different format for storing its value. Furthermore, there are slightly different versions for variables with single character names as opposed to longer names. Finally, we will also examine the structure of numerical and string *arrays*.

Getting back to our previous results, we can generalize the storage of any single-letter, numeric variable within the T/S 2068 as shown in Fig. 3–1.

The first byte equals the ASCII code for the letter which is the variable name. Fig 3–1 shows this in a slightly different fashion:

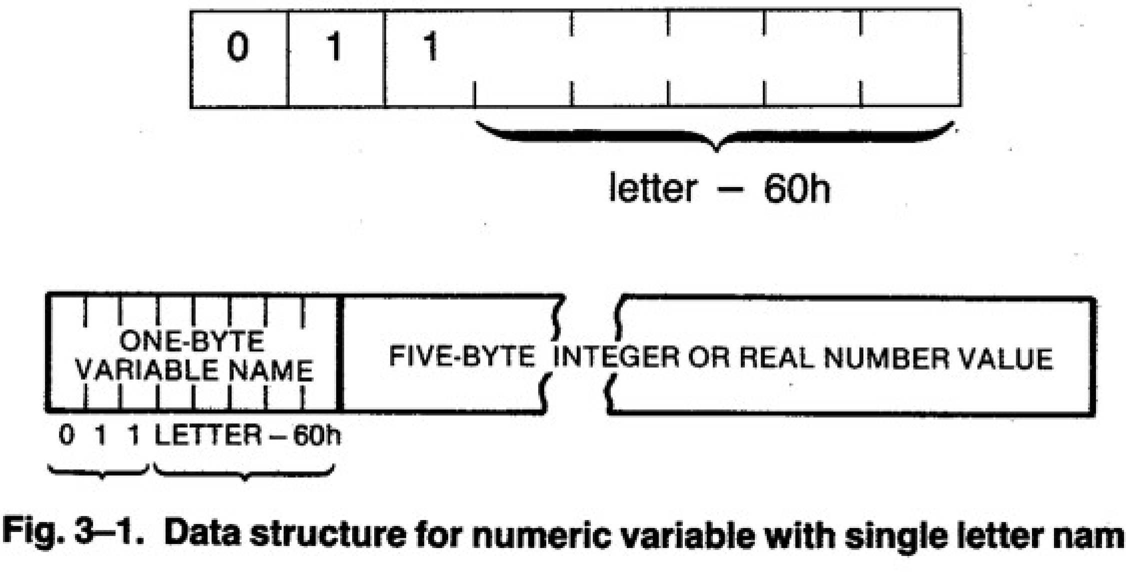


Fig. 3–1. Data structure for numeric variable with single letter name.

The reason for this format will become clear as we discuss the other types of variables. In this case, however, we can note that placing the three bits 011 in front of any 5-bit value is equivalent to adding 01100000 to that number. Converting to hex, we find that this means adding 60h. Therefore, adding 60h to "letter – 60h" just means that the letter is stored as is.

BYTE 1	BYTE 2	BYTE 3	BYTE 4	BYTE 5
0	SIGN	LSB	MSB	0
ALWAYS ZERO (SIGNIFIES INTEGER VALUE)	0 = POSITIVE FFh = NEGATIVE	16-BIT VALUE (LEAST SIGNIFICANT BYTE FIRST)		ALWAYS ZERO

Fig. 3–2. Data structure for an integer value (−65535 to +65535).

We will be using this technique of subtracting 60h from a letter's ASCII code, so it does deserve some explanation. Since a variable name must *always* begin with a letter, the first (or only) character in a variable name can only take on 26 different values. This is also due to the fact that the T/S 2068 treats upper-case and lower-case letters as if they were the same *(as far as variable names are concerned).* Since we only need 5 bits to describe 26 different items, the T/S 2068 steals the remaining 3 bits of the first character in a variable name for indicating what type of variable it is.

For any numerical variable (integer and real) the next five bytes hold the current value for that variable. In our example, the next byte is a zero and this indicates that the value is stored in integer format. Fig. 3–2 outlines the significance of each byte in an integer data element.

Following the zero is a byte that represents the sign of the integer. It will be zero if the number is positive, 255 (FFh) if the number is negative. The next two bytes represent the lower and then higher byte of a 16-bit value. In our example, if we write the two bytes together we would get a value of 01_{256} which, of course, is equal to the number 1. This agrees with the value that we previously assigned using the LET statement. In general, the higher byte must be multiplied by 256 and then added to the lower byte to determine its decimal value.

The fifth remaining byte of an integer variable will always be zero as confirmed by the PEEK 26715. Finally, there is the value 128, which is the last byte at which we originally PEEKed. This is used to signify the end of the variable list.

If we now type:

```
LET b = -2
```

then a second variable, 'b', should get added to the list. Check this by typing: (I promise after this we'll use a FOR . . . NEXT loop):

```
PRINT PEEK 26716        (98)         b
PRINT PEEK 26717        (Ø)          integer
PRINT PEEK 26718        (255)        negative
PRINT PEEK 26719        (254)      }
                                   }    -2
PRINT PEEK 26720        (255)      }
PRINT PEEK 26721        (Ø)          always zero
PRINT PEEK 26722        (128)        end of table
```

We've added some comments on the right to illustrate the results. As you can see, the 128 at location 26716 has been replaced by a 98 which is the code for the letter b. The next byte is zero, signaling that an integer value follows. The next byte is 255 which indicates that the number is negative. The following two bytes when taken together form the integer value. But 255 * 256 + 254 certainly does not equal 2. As you may have guessed, the T/S 2068 stores a negative number using its twos complement notation. Check for yourself that this value is indeed the correct result for a negative two. One way to determine the decimal value of a negative integer is to subtract 131,072 from its *unsigned* value (e.g., 255 × 256 + 254 = 131,070 − 131,072 = −2).

A good question to ask at this point might be: Why use five whole bytes to store a number that could be put into two bytes plus one extra bit? In fact, most other computers limit integers to one half of this range (i.e., −32768 to +32767) and thereby only require two bytes for each integer. The only possible answer is that it makes integer storage more compatible with that of floating point numbers. These are described next.

Floating Point

Many times we need to work with numbers larger than 65,535. While we could always add more bytes to represent numbers of any given size, it becomes much too inefficient as the number gets very large. Furthermore, we also need some way to signify fractions such as ½. The solution to both of these problems is to use *scientific* *notation*. Numbers stored this way are also called *floating point numbers.*

You may already be familiar with this technique from working in BASIC. Whenever the computer needs to display a very large, or very small number, it uses the form:

x.yyyyE±nn

Where x.yyyy represents the *mantissa* and nn is the *exponent.* Of course, we know that this is read as x.yyyy times ten to the nn power. You should be able to see that scientific notation lets us represent very, very large (or small) numbers, although only a few significant figures are retained.

The same procedure can be applied to writing binary numbers. Only now the exponent represents powers of two instead of ten. If we assign one byte to represent the exponent, then we would have a range of 0-255. We also need to consider negative exponents. A convenient way to deal with this is to add 128 to the exponent before storing it and then subtract it again to retrieve the correct number. Then the exponent can have a value between −128 and +127. Raising two to these powers gives us an equivalent range of about 10E−30 to 10E+30. This should certainly prove adequate for most jobs.

Therefore, whenever a number is too large to be stored as an integer, or if it contains some fraction, binary scientific notation is used. First the number must be *normalized* which means converting it to a mantissa between ½ and 1. (This is equivalent to how we normalize decimal numbers in scientific notation — where we require the number on the left of the decimal point to be between 0 and 10). If the mantissa is not already in this range, it is multiplied or divided by 2 as many times as necessary until it is. Each time this is done, the exponent is decreased or increased by one, to keep track of the correct value. With the mantissa normalized and the correct exponent calculated, we are almost ready to determine how it will be stored in memory.

First of all, we can arbitrarily decide how many bytes to use for the mantissa. Many numbers will have an infinitely long mantissa just like the decimal form of ⅓ is .3333 . . . Therefore, many numbers will require that the mantissa be rounded off to a fixed number of digits. In the T/S 2068, floating point numbers are stored with a 4-byte, or 32-bit mantissa. Thirty-two bits lets us represent numbers as high as

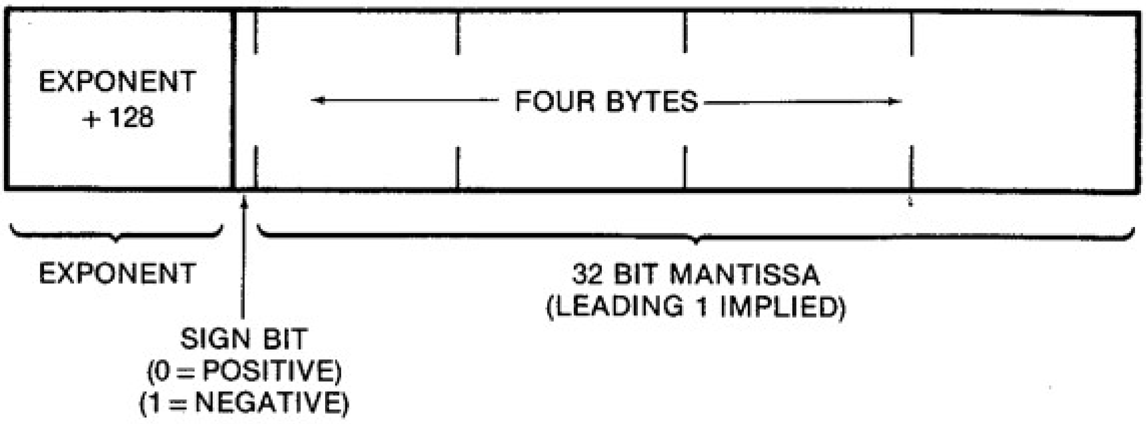


Fig. 3–3. Data structure for floating point value.

4,294,967,296 which means that we will have the equivalent of over 9 decimal-place accuracy.

There is one more point to consider and that is the overall sign of the number. We know that the sign of the exponent has been accounted for by adding 128 to it. It turns out that the first bit in the mantissa will always be a one (since the mantissa has been normalized to a number great than ½ or 0.1 in binary). Therefore, this bit really does not need to be stored. In its place, we can then put a sign bit — 0 for positive, 1 for negative. At last, we can show the general form for storing a floating point number (see Fig. 3–3).

To check this on the computer, type:

```
LET c = 1/10
```

and then:

```
FOR i = 26722 TO 26727: PRINT i , PEEK i: NEXT i
```

Notice that we type this entire line as an immediate command (i.e., without a line number). By now you should understand why. Your screen should now look like this:

```
26722          99          c
26723          125         −3 (+128)
26724          76
26725          204
26726          204           1100 . . .
26727          205
```

To understand these numbers, let's calculate what ⅒ should be in binary notation. Referring back to Chapter 2, we can calculate that:

⅒ = Ø.ØØØ11ØØ11ØØ11ØØ11ØØ . . .

Normalization of the mantissa is really easy to perform. Simply count how many places to the right of the binary point we must go to reach a 1. In this case, there are three so the exponent will be 128 − 3 = 125. Moving the binary point over, we now have a mantissa of:

.11ØØ11ØØ11ØØ11ØØ . . .

Because we only have 32 bits, the mantissa is rounded off to that many places. Because the 33rd number is a one, the 32nd bit gets rounded up from 0 to 1. Finally, we drop the leading 1 and replace it with the sign bit, indicating that the number is positive. Thus ⅒ would be stored as:

```
Ø11111Ø1  Ø1ØØ11ØØ  11ØØ11ØØ  11ØØ11ØØ  11ØØ11Ø1
```

If we now convert each byte to its decimal value, we will come up with the exact same results printed by the computer.

We'll now examine what happens when we add a variable whose name is more than one character. Type ENTER to clear the screen and then:

```
LET dog = 3
```

Then:

```
FOR i=26747 to 26754: PRINT i, PEEK i: NEXT i
```

Your screen should now look like this:

```
26747          164          d (+64)
26748          111          o
26749          231          g (+128)
2675Ø          Ø           ⎫
26751          Ø           ⎪
26752          3           ⎬  integer value 3
26753          Ø           ⎪
26754          Ø           ⎭
```

Once again we will start with the general form for a numeric variable whose name is longer than one letter. This form is shown in Fig. 3–4.

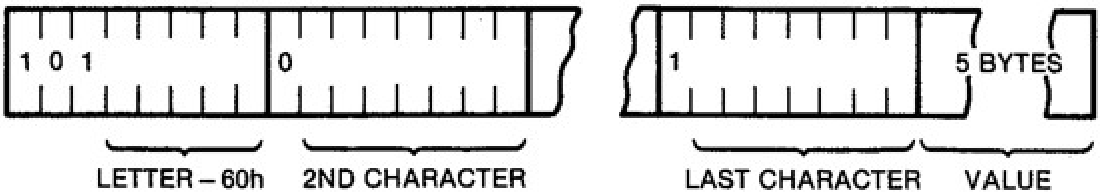


Fig. 3–4. Data structure for numeric variable with multiple character name.

The most important difference here is in the way the first letter is stored. The first three bits have been changed from 011 to 101. This is the signal that the T/S 2068 uses to distinguish between single and multicharacter variable names. Subtracting 60h from a character and then adding 101 in front of it is the same as adding 64 to the ASCII code. After placing the first letter in memory in this fashion, the remaining letters or numbers in the name are added in sequence. The normal 7-bit ASCII code for each additional character is used with the eighth, most significant bit set to zero. Since variable names in the T/S 2068 can be of any length, we will also need some way to determine which byte holds the last character of the variable name. For this purpose, that eighth bit proves to be most convenient. By setting it to a one, the computer can indicate that this is the end of the variable name and that the value follows. The next five bytes then contain the integer or floating point value, as previously described.

Numeric Arrays

There is one more type of numeric variable to consider and that is single or multidimensional *arrays*. To store an array, The T/S 2068 starts as usual with the variable name (array names are always a single letter). This time, the first three bits are set to 100, indicating that the following bytes hold information about an array. This is equivalent to storing the code for the letter plus 32. Next come two bytes which tell the computer how many more bytes follow that are associated with this array. The next byte holds the number of dimensions, or how many subscripts are used to identify a given element in the array. Next follow pairs of bytes, specifying the size of each dimension as declared when the array was DIMensioned. After storing the last dimension size, the actual elements follow, each using five bytes to store a value in either integer or floating point format. All of this is shown in Fig. 3–5.

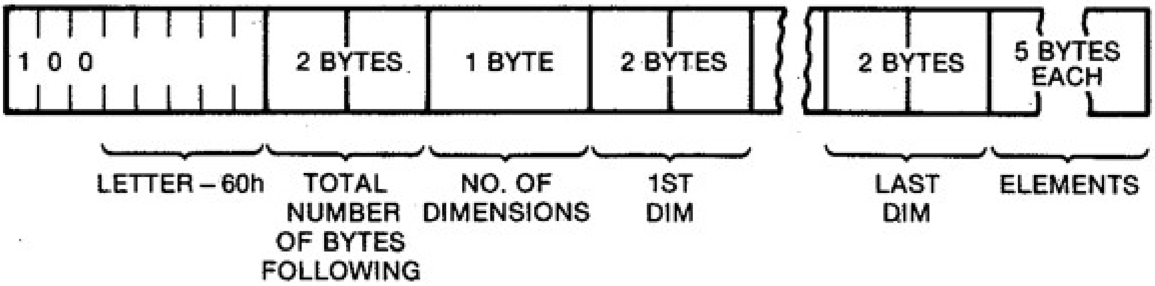


Fig. 3–5. Data structure for numeric array.

The order in which the elements are stored can best be viewed as removing the commas from the subscript and then going in ascending numerical order. For example, the elements of a 2×3 array called 'b' would be stored in the order:

b(1,1)b(1,2)b(1,3)b(2,1)b(2,2)b(2,3)

Here is a little test to demonstrate the storage of an array. Type ENTER so that the screen will clear. Then type:

```
DIM e(2)
LET e(1,1)=1
LET e(1,2)=2
LET e(2,1)=3
LET e(2,2)=4

FOR i=26755 TO 26782: PRINT i, PEEK i: NEXT i
```

This will yield the following results (you will have to hit ENTER to scroll the display):

```
26755          133        e (+32)
26756          25         } Number of bytes
26757          0          }  to follow
26758          2          Number of dimensions
26759          2          } First dimension
26760          0          }
26761          2          } Second dimension
26762          0          }
26763          0          }
26764          0          }
26765          1          }  1  e (1,1)
26766          0          }
26767          0          }
```

```
26768          Ø   }
26769          Ø   }
26770          2   }  2  e (1,2)
26771          Ø   }
26772          Ø   }
26773          Ø   }
26774          Ø   }
26775          3   }  3  e (2,1)
26776          Ø   }
26777          Ø   }
26778          Ø   }
26779          Ø   }
26780          4   }  4  e (2,2)
26781          Ø   }
26782          Ø   }
```

String Variables

Now that we understand how numeric variables are stored in memory, it will be quite easy to describe string variables. Another simplification is that all string variables use single letter names. Thus we have only two more *data structures* to consider: simple string variables and string arrays. A simple string variable uses the form shown in Fig 3–6.

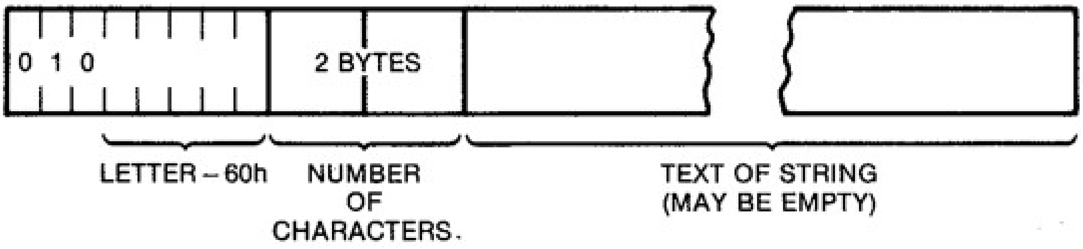


Fig. 3–6. Data structure for string variable.

Notice, again, that the first three bits of the name take on a unique pattern so that the computer can tell that this is a string variable. Two more bytes are then necessary to specify the length of the string. Following that, we have the actual characters in order as they appear in the string. Simple, isn't it? To test this out type:

```
LET f$ = "Test"
```

Then:

```
FOR i=26783 TO 26789: PRINT i, PEEK i: NEXT i
```

which will yield:

```
26783          70          f (-32)
26784          4           } Number of
26785          0           } characters
26786          84          T
26787          101         e
26788          115         s
26789          116         t
```

String Arrays

Finally, we come to string arrays which are stored as shown in Fig. 3–7. To verify this, type:

```
DIM g$(3,3)
LET g$(1)="one"
LET g$(2)="two"
LET g$(3)="three"

FOR i=26790 TO 26807: PRINT i, PEEK i: NEXT i:
```

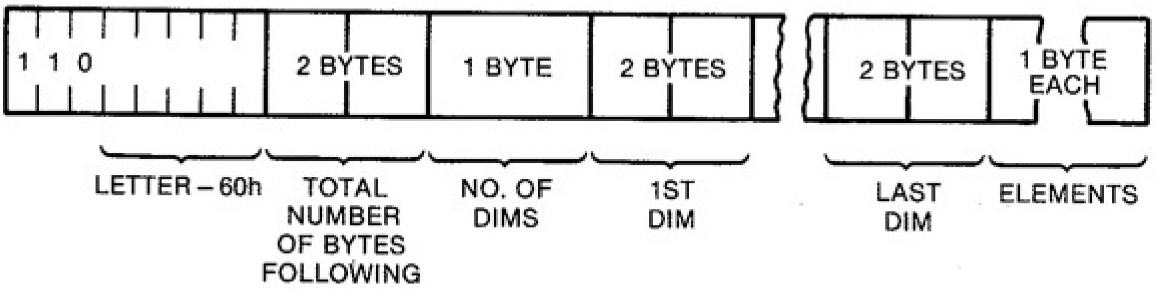


Fig. 3–7. Data structure for string array.

This will give:

```
26790          199          g (+96)
26791          14           } Number of bytes
26792          0            } to follow
26793          2            Number of dimensions
26794          3            }
26795          0            } First dimension
26796          3            } Second dimension
26797          0            }
26798          111          o
26799          110          n
26800          101          e
26801          116          t
```

26802	119	w
26803	111	o
26804	116	t
26805	104	h
26806	114	r
26807	128	end of list

Notice that the last byte is 128, which is the flag to signal the end of the variable list. This shows that the string "three" has been truncated to "thr" because we DIMensioned g$ for a length of 3. If we were to lay out the current variable list it would look like Fig.. 3–8.

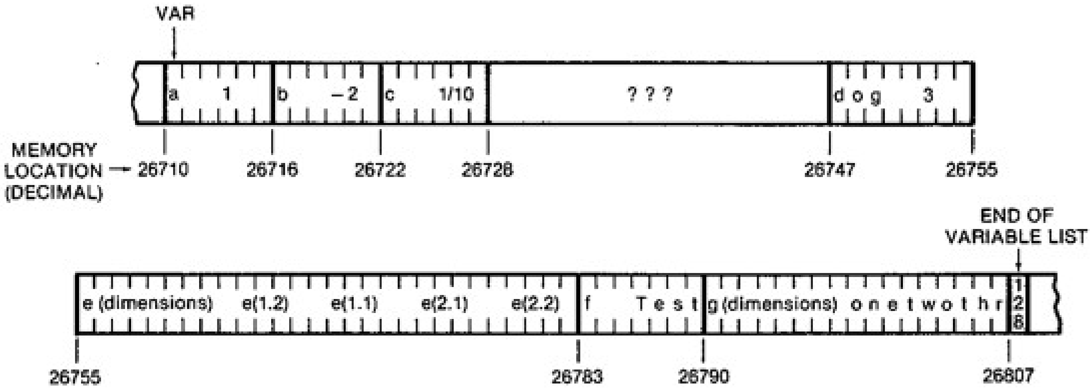


Fig. 3–8. Typical variable list.

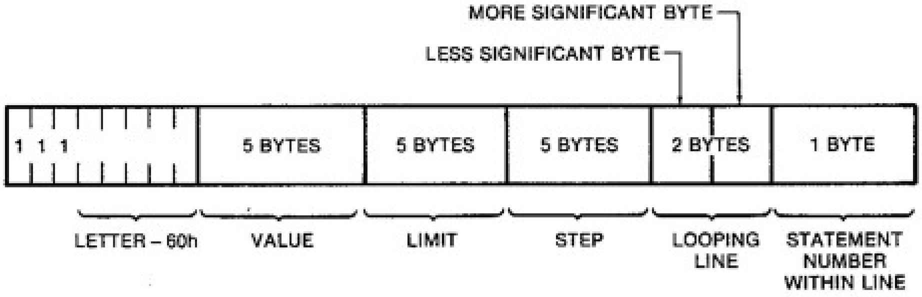


Fig. 3–9. Data structure for FOR . . . NEXT loop variable.

FOR . . . NEXT Loop Variable

Did you notice that we skipped a block of memory between locations 26728 and 26746? It is this area that we want to explore now. We will find that it contains information about a special type of variable: the control variable in a FOR . . . NEXT loop (e.g., 'i' in FOR i = 1 to 10).

Remember when we got tired of typing in PEEK statements one at a time? After defining the variable called 'c', we used a FOR . . NEXT loop to print out the memory locations occupied by this variable. As we did this, the computer had to define a new variable called 'i', to keep track of the loop statement. It did this by adding another entry to the variable list in the form shown by Fig. 3–9. Thus if we type:

```
FOR i=26728 TO 26746: PRINT i, PEEK i: NEXT i
```

we should see:

```
26728          233          i (+128)
26729          Ø
26730          Ø
26731          1Ø7              26731 (current value)
26732          1Ø4
26733          Ø
26734          Ø
26735          Ø
26736          122              26746 (limit)
26737          1Ø4
26738          Ø
26739          Ø
26740          Ø
26741          1                1 (step)
26742          Ø
26743          Ø
26744          254              looping line
26745          255
26746          2            statement number
```

These results should be self explanatory except for the "current value." Why does it equal 26731? That is certainly not the value of 'i' after the loop has finished. (It will be 26747 since a control variable must always go one step beyond the control limit value in order to exit the loop). This value does make sense, however, when you consider what the value for 'i' was during the particular pass through the loop when this was printed. To PEEK at memory location 26731, the variable i had to equal this value at that point. Therefore, when the loop was printing out the value at location 26731, the variable 'i' equaled 26731 which required that a value of 107 be at that address.

Since we entered the command without a line number, the two bytes that represent the "looping line" contain a value of 65534. This, of course, would not be a valid line number in BASIC. The *statement* number, however, does show a correct value of two. This means that after each pass through the loop (i.e., when we reach the NEXT i statement) the computer should return to the second statement in the command line. This takes us back to the statement following the first colon, which was the PRINT command.

Varying variables

As the value for any variable changes, it will normally stay in its same position within the variable table. This is true even if the value changes from an integer number to a floating point. For example, type:

```
LET dog = .1
```

Then:

```
FOR i=26747 TO 26754: PRINT i, PEEK i: NEXT i
```

This will give:

```
26747        164        d (+64)
26748        111        o
26749        231        g (+128)
26750        125
26751        76
26752        204          real value 0.1
26753        204
26754        205
```

Compare this with our earlier results. Notice that the variable *dog* now has a new value which is in floating point format. You can also see that the value is identical to that of the variable c. This, of course, just proves that ⅒ is equivalent to .1

There is one exception to the rule of variables staying in the same place. This comes about because we allow simple string variables to change in length as they are assigned new values. For example, let's type:

```
LET f$ = "Examination"
```

There is no way for the computer to replace the old value with the new one at the same location. This is because we only have four bytes holding the old value "Test" and the new value, "Examination" requires 11 bytes. We can't just add on a few more bytes because that would begin to run into the next variable on the list, in effect, wiping it out. The only solution is to create a new variable entry for f$ at the end of the list.

But what about the old entry for f$? It would certainly be confusing to have more than one entry in the list for the same variable. There are basically two alternatives to solving this problem. One method would be alter the first entry somehow so that it no longer looks like a valid variable entry (i.e., "mark" it as being unused or garbage). The other method would involve removing the entry completely and then moving the entire rest of the list up to fill in the void left by the old entry. It is interesting to note that the BASIC interpreters in most personal computers use the first method. This offers some advantages but has one major drawback. By leaving the old entries still in memory, a program can create a growing trail of garbage which takes up valuable memory space with useless information. If the program eventually uses up all of the available RAM, it must then go through a process known as "garbage collection" whereby all of these unnecessary entries are purged and a new clean list is created.

Unfortunately, this process can take a considerable amount of time during which the normal program execution is put on hold. Many a novice programmer has been baffled when trying to figure out why his/her program randomly seems to hang up for a few seconds (or even minutes!) but then continues as if nothing had happened. While this problem is somewhat rare (many people never experience it), the chances of garbage collection occurring increase with programs that constantly re-define string variables and with the large programs because large programs take up more memory, leaving less for variable storage.

To see what has happened on the T/S 2068, type:

```
FOR i=26783 TO 26789: PRINT i, PEEK i: NEXT i
```

showing:

```
26783              199              g (+96)
26784              14               } Number of bytes
26785              Ø                }  to follow
26786              2
26787              3                } first dimension
26788              Ø
26789              3                etc.
```

Where the old variable f$ had been, we can now see the beginning of g$. That is, the old entry for f$ has been removed and the next variable (in this case, g$), as well as everything else in the list, has moved up. Thus the T/S 2068 does its garbage collection "on the fly" every time a string variable changes length. To find the new location for f$, we can calculate 26807 (previous end of list) − 7 (number of bytes used in old f$ entry) = 26800. This should be the beginning address for the new f$ entry.

Therefore type:

```
FOR i=26800 TO 26814: PRINT i, PEEK i: NEXT i
```

to get:

```
26800          70          f (−32)
26801          11          }  new length
26802          0           }
26803          69          E
26804          120         x
26805          97          a
26806          109         m
26807          105         i
26808          110         n
26809          97          a
26810          116         t
26811          105         i
26812          111         o
26813          110         n
26814          128         end of list
```

A map of the variable list would now look like Fig. 3–10. Compare this to Fig. 3–8.

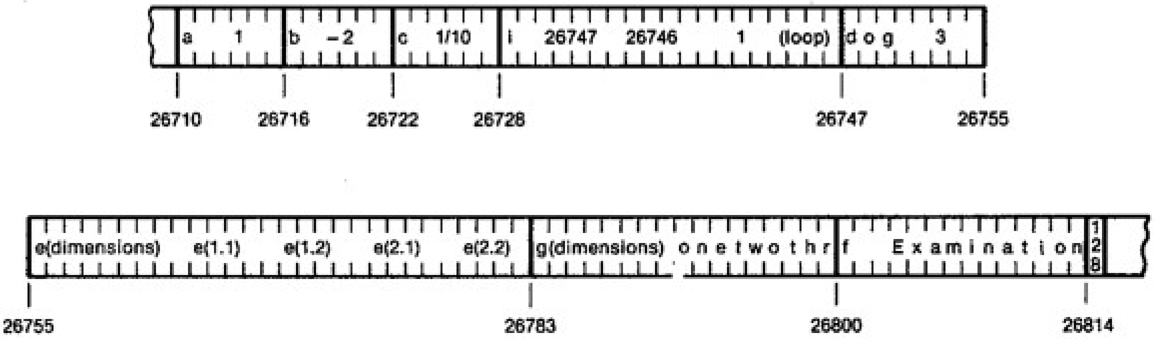


Fig. 3–10. Variable list after string variable is changed.

HOW THE BASIC INTERPRETER OPERATES

We've come a long way in describing how the T/S 2068 BASIC interpreter stores data in memory. Being able to reference the information using symbolic names (even if they're only one character long) is much easier than keeping track of the absolute memory locations needed by the CPU. Of course, this is just a tiny part of what the interpreter does.

Before leaving the subject of the variable list, we must describe one more thing: how the interpreter finds a given variable in the list. This turns out to be quite trivial, since we have satisfied the two requirements that we mentioned at the beginning. That is, knowing where the list starts and where each variable description ends and the next one begins. This first requirement was taken care of by keeping a pointer in the reserved RAM. These locations (23627, 23628) always hold the starting address for the first variable in the list. As we move through the list, we can determine the boundaries using a few simple rules:

- Simple, single character variables always take up 6 bytes.
- String variables can be of any length but the actual length is always stored in the 2 bytes following the variable name.
- Arrays also contain 2 bytes following the name, to indicate how many more bytes are used.

Therefore, whenever the BASIC interpreter is asked to find the value of a given variable, it can make a *linear search* through the variable list. This means that it starts at the top of the list and checks the name of the desired variable with the name of the first entry. If these do not match, the interpreter calculates where the next variable begins and then checks its name. As long as there is no match, the interpreter will keep hopping down the list, checking each entry. If it eventually finds a match, then the value can be read out (or a new value put in). If the end of the list is reached without a match, then a "variable not found" message is displayed. If, however, the command is allowed to create a new variable (e.g., LET, FOR, INPUT, etc.), and no such entry already exists, then a new entry is made at the end of the list.

It is important to note that variables in the list are kept in the order that they were created. This gives us an important clue for speeding up BASIC programs. Since the interpreter will have to perform a search of the list each time a variable is referenced, those variables that are used most often should be defined first. They then will be at the top of the list for quick access by the interpreter.

This completes our discussion of how variables are stored in memory. We are now ready to investigate how BASIC programs themselves are stored in memory.

PROGRAM STORAGE

If you've followed everything so far, then you will have no trouble understanding the way BASIC stores *programs* in memory. Compared to the ways in which variables are stored in memory, program storage is much simpler. In fact, every line of a BASIC program is stored in the same fashion. The general form is shown in Fig. 3–11.

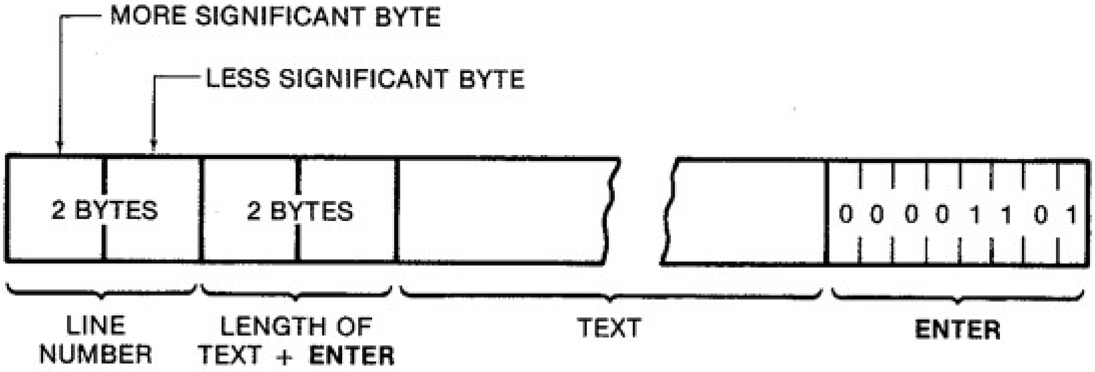


Fig. 3–11. Data structure for a BASIC program line.

As we mentioned before, whenever you hit the ENTER key, the preceding characters are examined by the parser. If there are no errors and the first character on the line is a number, then the parser determines that this is a BASIC program line. The interpreter then places the line into a special area of memory set aside for the storage of BASIC program lines.

Actually, the BASIC program is stored in memory in much the same way as variables. To begin with, each line is stored consecutively in what we could call the BASIC program list. The beginning address for the list is kept in a pointer called PROG (within the reserved area of RAM). Instead of variable names, we keep track of program lines by their *line number*. Since each program line can be anywhere from 1 to 65K characters long, we add two bytes to specify its length. This is identical to what we did with string variables. The only major difference with the way program lines are stored is that they are always kept in numerical order according to line number. Thus if the program has line numbers, 10, 20, 30, and 40 and we add a line number 25, it gets sandwiched in between 20 and 30. Of course, you probably have seen this whenever you LIST a program.

Let's take a closer look at how the text of our BASIC program lines are kept in memory. For the most part, the text is stored exactly as it appears on the screen. One exception is that key words (PRINT, NEXT, GO TO, etc.) are stored using a single byte, or *token*. This saves considerable memory space over storing each character in the key word as a single byte. These tokens are part of the character set as shown in Table 2–5.

To start our investigation using the computer, type:

```
NEW
```

so that we're all at the same starting point. Now type:

```
10 REM hello
```

With a program line entered into memory, the next step is to find the starting address of the program image. We know this will be found at locations 23635-6 so we type:

```
PRINT PEEK 23635 + 256 * 23636
```

The answer comes up 26710. With this information, we can now type:

```
FOR i=26710 TO 26720: PRINT i , PEEK i: NEXT i
```

This displays on the screen:

```
26710          0         } 10 (line number)
26711          10        }
26712          7         } 7 (length of text)
26713          0         }
26714          234       REM token
26715          104       h
26716          101       e
26717          108       l
26718          108       l
26719          111       o
26720          13        ENTER
```

A couple of notes are in order here. First, notice that the line number bytes are stored with the most significant byte first. This is the opposite of most other two-byte storage. Next, note that the line count includes the ENTER key (and it is stored in memory). Another new technique used in program line storage concerns numerical constants. If we type:

```
    2Ø LET a = 2
```

and then:

```
    FOR i=26721 TO 26735: PRINT i, PEEK i: NEXT i
```

we get:

```
    26721            Ø          }  2Ø (line number)
    26722            20         }
    26723            11         }  11 (length of line)
    26724            Ø          }
    26725            241        LET token
    26726            97         a
    26727            61         =
    26728            5Ø         2
    26729            14         number token
    2673Ø            Ø          }
    26731            Ø          }
    26732            2          }  2
    26733            Ø          }
    26734            Ø          }
    26735            13         ENTER
```

As you can see from the notations, whenever we have a numerical constant in a BASIC line, it is followed by a special token byte (with the value 14) and then by five more bytes with the constant's value in integer or floating point format.

There you have it — program storage on the T/S 2068. We can now describe some of the functions performed by the BASIC interpreter. We've already seen how the interpreter keeps a variable list and a program list in memory. As we, or our program, use new variable names, they are added to the variable list. If we enter BASIC program lines, they are added to the program list.

If we type LIST, we know that the interpreter will display the current program in memory. It does this by simply going to the start of the program list and printing out the text that it finds. Double-byte line numbers are converted to their decimal form and printed on the screen. Tokenized keywords are also expanded into their English-like form.

Explaining what happens when you type RUN is a little more complicated. In fact, it is a *lot* more complicated. The basic operation, however, is to read each program line in sequence and convert the keyword commands into some appropriate action. For this purpose these are numerous subroutines stored in ROM that perform each of these tasks. Therefore, as a program line is read by the interpreter, it looks at the token to decide what subroutine(s) to execute. If further data is needed, such as variables or constants, they are read from the program line and sent to the subroutine. When a branching instruction, such as a GO TO 20, is reached, the interpreter looks down the program list for the appropriate line number and then resumes execution at that point. That brings us to helpful hint number two.

When writing long programs, it will speed things up if you place often used subroutines at the beginning of your program. Such routines, which may be called from various places throughout the program, will execute faster since the interpreter will not have to search through the entire program list looking for them. Of course, if you really want to speed up your programs, then machine language may be the only answer. That's the subject of Part Two. First, however, we need to know a little bit more about the T/S 2068 hardware. In particular, we shall start with the Z80 CPU which is described in the next chapter.

SECTION B
INSIDE THE T/S 2068

4
The Z80 CPU

INTRODUCTION

Our discussion of what's inside the T/S 2068 begins with a detailed look at the Z80 CPU. This chapter contains a lot of information which may interest only the "hardware hacker" who wants to know the details of the Z80's operation. Much of this material will also be explained in later chapters. If you feel that things are getting a little too deep, skip on to the next section. The Z80 is a very complicated piece of silicon as you can see from the photomicrograph of Fig. 4–1. Fortunately, we don't have to know what's inside a Z80 to be able to program it. But a knowledge of its *architecture* is essential.

Fig. 4–2 shows a block diagram of the Z80 CPU. The arrows indicate how each section is interconnected and how the CPU communicates with the rest of the computer. These arrows indicate how the Z80 is organized into several buses. A *bus* is simply a group of wires that are treated collectively. For example, all data flowing into or out of the CPU pass along eight wires called the *data bus*. For this reason, the Z80 is referred to as an 8-bit CPU.

When data is transferred to or from main memory, a 16-bit *address bus* is used to specify which memory location to use. This gives the Z80 the capability of directly addressing 64k of memory. (The T/S 2068 can actually accommodate much more memory as we will see in the next chapter.) Within the Z80, data is stored in special memory cells called *registers*. These registers are also connected to the Arithmetic and Logic Unit, or ALU, where various operations are performed on the data. Finally, there is an instruction decode and CPU control section which interprets the machine language program instructions and tells the rest of the CPU what to do.

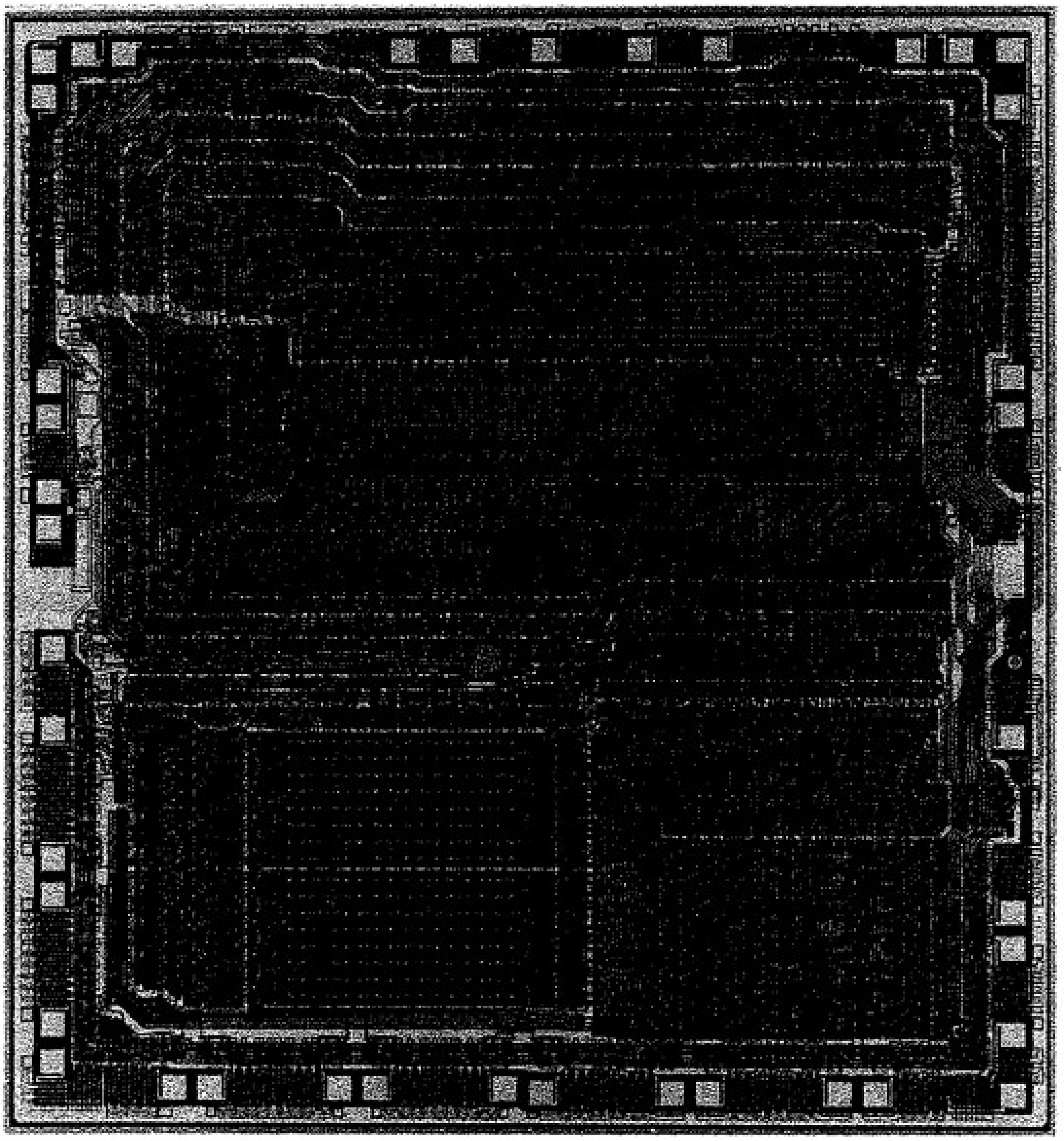


Fig. 4–1. Photomicrograph of Z80 CPU. (*Courtesy Zilog, Inc.*)

REGISTERS

The Z80 contains 208 bits of RAM that are available to the programmer. Fig. 4–3 illustrates how this memory is organized into eighteen 8-bit registers and four 14-bit registers. The *special-purpose* registers hold information that is closely related to the Z80 hardware and its operation. The general-purpose registers are used by the programmer for temporary storage of data that is to be processed by the CPU.

Special-Purpose Registers

Program Counter (PC)—The program counter holds the 16-bit address of the current instruction byte being fetched from memory. A

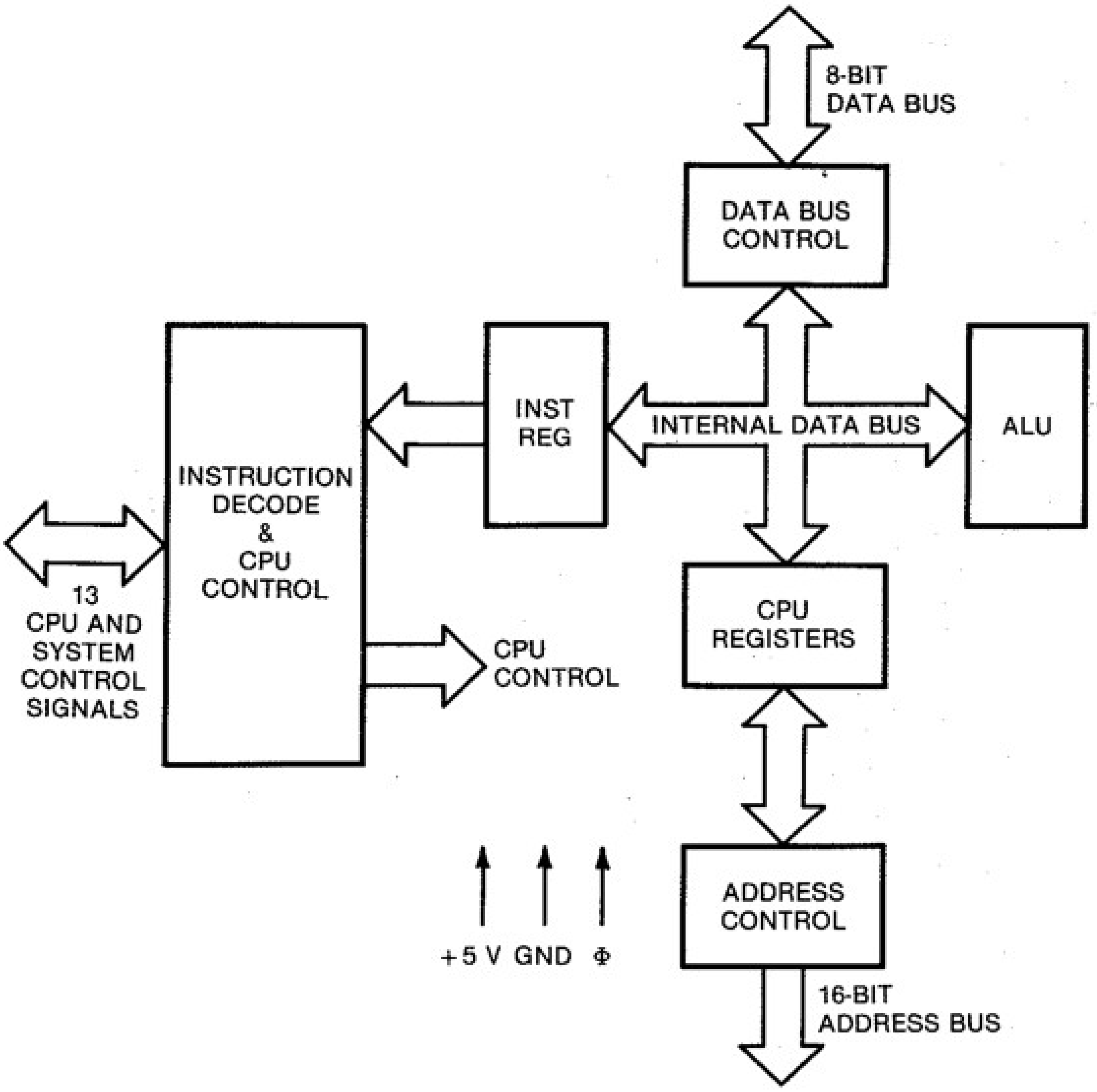
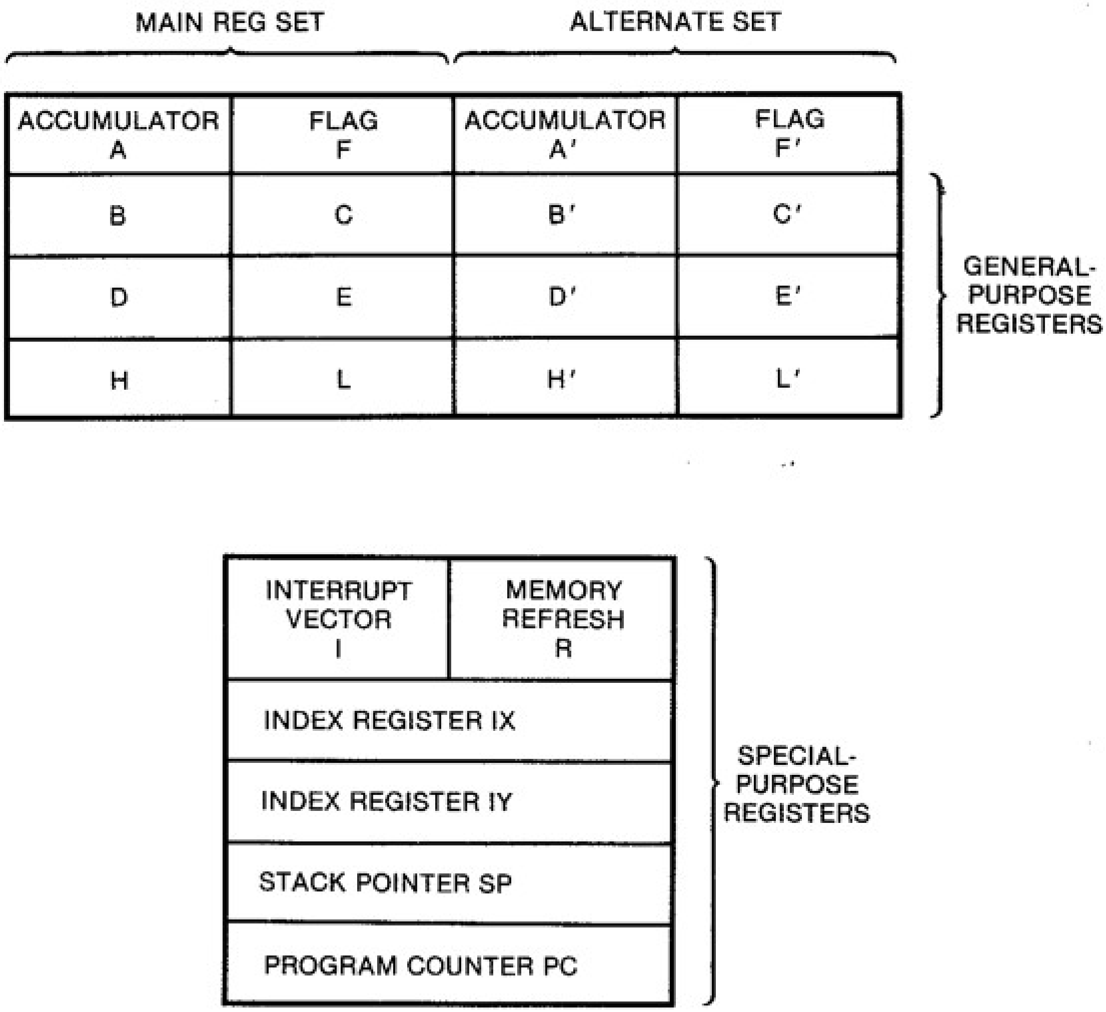


Fig. 4–2. Block diagram of Z80. (*Courtesy MOSTEK Corp.*)

special circuit in the Z80 automatically increments this counter after each byte is read. This is what causes the CPU to execute programs normally in sequential order. If a jump or call instruction is encountered (equivalent to the BASIC GO TO and GO SUB statements), then the new address is placed in the PC register.

Stack Pointer (SP)—The stack pointer holds a 16-bit address which points to a section of system RAM that is reserved for the machine stack. The stack is a special type of memory organization whereby data can be stored and retrieved on a last-in, first-out (LIFO) basis. We'll explain this further in Chapter 10.

Index Registers (IX and IY)—Each of these two registers can hold a 16-bit value. While they can be used as general-purpose storage, their primary function is to hold the base address for a special type of



ACCUMULATOR A	FLAG F	ACCUMULATOR A′	FLAG F′
B	C	B′	C′
D	E	D′	E′
H	L	H′	L′

INTERRUPT VECTOR I	MEMORY REFRESH R
INDEX REGISTER IX	
INDEX REGISTER IY	
STACK POINTER SP	
PROGRAM COUNTER PC	

Fig. 4–3. The Z80 register set. (*Courtesy MOSTEK Corp.*)

memory access known as indexed addressing. This will also be explained in Chapter 10.

Interrupt Vector (I) and Memory Refresh (R)—These are two 8-bit registers which support some of the advanced features of the Z80. The I register is used with an advanced interrupt handling mode available on the Z80. When this mode is selected, the I register supplies the high order address for an indirect call to the interrupt service routine. The R register is used to refresh dynamic RAMs automatically.

Accumulator and Flag Registers

The Z80 has two independent 8-bit accumulators, each with a respective 8-bit flag register. These are also referred to as the A and F registers, or sometimes treated together as a single 16-bit AF register. The programmer can alternate between the two accumulator and flag pairs by executing a single exchange instruction.

The accumulator is used to hold the results of 8-bit arithmetic and logical operations. It is also supported by more instructions and addressing modes than any other register. The flag register contains six special bits which indicate vital statistics about the value in the accumulator and the results of the last operation. These bits can be "tested" by other instructions to allow conditional program execution based on the results of a previous operation.

General-Purpose Registers

There are two matched sets of general-purpose registers in the Z80. Each set contains six 8-bit registers that can be used individually or treated as 16-bit register pairs. One set is called the BC, DE and HL registers, while the other set is called BC', DE', and HL'. Just as with the duplicate AF registers, the programmer can exchange all three register pairs with their "primed" counterparts using a single instruction.

THE FLAG REGISTER

The notion of a *flag register* is very important to the design of a CPU. Therefore, we should examine the Z80's flag registers in a little more detail. Although there are actually two separate F registers, only one is used at a time. Therefore, we will continue the discussion as if there were only one flag register.

As we have already said, six of the eight bits in the flag register are used to denote specific details about the operation of the CPU. The remaining two bits are unused. Four of these six bits are testable — that is, they can be used as a condition for the execution of certain instructions. These are the Carry flag (C), Zero flag (Z), Sign flag (S), and Parity/Overflow (P/V) flag. Their bit positions in the Flag register are shown next:

D7							DØ
S	Z		H		P/V	N	C

Testable Flag bits: C, P/V, Z, S

The Carry flag is used to indicate that an arithmetic operation has exceeded the 8-bit range available to the accumulator. For example, adding the following two numbers would cause a carry to be generated:

```
             1Ø1ØØØ11
         +   11ØØ1ØØØ
         ------------
      C
    = 1      Ø11Ø1Ø11
```

When the carry flag equals 1, it shows that a carry has occurred. The carry flag will also be set if a *borrow* occurs during a subtraction operation. Several other instructions, such as the shift and rotate group, also affect this bit.

The Parity/Overflow flag serves a dual purpose, depending upon the type of operation being performed by the CPU. When logical operations are performed (such as AND A,B), it indicates the parity of the result in the accumulator. Parity is derived from the number of "1" bits in a given byte and can be either odd or even. After performing an arithmetic operation (such as ADD A,B), the P/V flag indicates whether there has been an overflow condition. This happens when signed numbers are used and result of the operation cannot be represented correctly by the 8-bit accumulator.

The Zero flag is set whenever the results of an operation leave a value of zero in the accumulator. It is also used in a slightly different fashion by the block I/O and search instructions as well as the BIT testing instruction.

The Sign Flag is a replica of the most significant bit of the accumulator. When signed numbers are being manipulated by the CPU, the most significant bit represents the sign of the number. A "0" indicates that the accumulator holds a positive number, while a "1" shows that it is negative.

Nontestable Flag Bits: H and N

The other two bits in the flag register are used to implement BCD arithmetic. The Half carry (H) flag keeps track of any carry or borrow between the lower four bits and the upper four bits of the accumulator. The Add/Subtract (N) flag indicates which type of operation was performed last. These two bits are used by the DAA (Decimal Adjust Accumulator) instruction to properly re-format the contents of the accumulator into packed BCD format. This allows the Z80 to perform

Table 4–1. Summary of Flag Operations (Courtesy MOSTEK, Corp.)

	D7							D0	
						P/			
Instruction	S	Z		H		V	N	C	Comments
ADD A,s; ADC A,s	↕	↕	X	↕	X	V	0	↕	8-bit add or add with carry
SUB,s; SBCA,s; CP,s; NEG	↕	↕	X	↕	X	V	1	↕	8-bit subtract, subtract with carry, compare and negate accumulator
AND s	↕	↕	X	1	X	P	0	0	} Logical operations
OR s; XOR s	↕	↕	X	0	X	P	0	0	
INC s	↕	↕	X	↕	X	V	0	●	8-bit increment
DEC s	↕	↕	X	↕	X	V	1	●	8-bit decrement
ADD DD, SS	●	●	X	X	X	●	0	↕	16-bit add
ADC HL, SS	↕	↕	X	X	X	V	0	↕	16-bit add with carry
SBC HL, SS	↕	↕	X	X	X	V	1	↕	16-bit subtract with carry
RLA; RLCA; RRA; RRCA	●	●	X	0	X	●	0	↕	Rotate accumulator
RL s; RLC s; RR s; RRC s; SLA s; SRA s; SRL s	↕	↕	X	0	X	P	0	↕	Rotate and shift locations
RLD; RRD	↕	↕	X	0	X	P	0	●	Rotate digit left and right
DAA	↕	↕	X	↕	X	P	●	↕	Decimal adjust accumulator
CPL	●	●	X	1	X	●	1	●	Complement accumulator
SCF	●	●	X	0	X	●	0	1	Set carry
CCF	●	●	X	X	X	●	0	↕	Complement carry
IN r, (C)	↕	↕	X	0	X	P	0	●	Input register indirect
INI; IND; OUTI; OUTD	X	↕	X	X	X	X	1	X	} Block input and output
INIR; INDR; OTIR; OTDR	X	1	X	,X	X	X	1	X	Z = 0 if B ≠ 0 otherwise Z = 1
LDI; LDD	X	X	X	0	X	↕	0	●	} Block transfer instructions
LDIR; LDDR	X	X	X	0	X	0	0	●	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI; CPIR; CPD; CPDR	↕	↕	X	↕	X	↕	1	●	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	↕	↕	X	0	X	IFF	0	●	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	X	↕	X	1	X	X	0	●	The state of bit b of location s is copied into the Z flag

The following notation is used in this table:

SYMBOL	OPERATION
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract. H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
	The flag is affected according to the result of the operation.
●	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care".
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction.
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>
nn	16-bit value in range <0, 65535>

BCD arithmetic almost as easily as binary arithmetic. Table 4–1 shows a summary of how the flag register is affected by various Z80 instructions. The importance of this will become clearer when we begin to examine the Z80 instruction set in Chapter 9.

CPU OPERATION

All CPU's operate by repeating a few basic steps over and over. These steps are performed according to a complex interconnection between various circuits within the CPU. These circuits operate "synchronously"; that is, they all perform their functions at a very specific time. All of the circuits within a Z80 are timed in accordance with a single clock signal. In the T/S 2068, this is a 3.528 MHz square wave which is derived from the 14.112 MHz master oscillator inside the computer. Each cycle of the CPU clock causes a specific action to take place within the Z80. This is also referred to as a *T cycle*. It takes several T cycles to perform a *machine cycle*. Each machine cycle represents the transfer of one byte into or out of the CPU. An *instruction cycle* consists of one or more machine cycles and represents the smallest unit available to the machine language programmer. Fig. 4–4 shows the basic CPU timing for an INC (HL) instruction. This operation involves reading the value of a memory location into the CPU, adding one to this value, and then storing the new value back into the memory location.

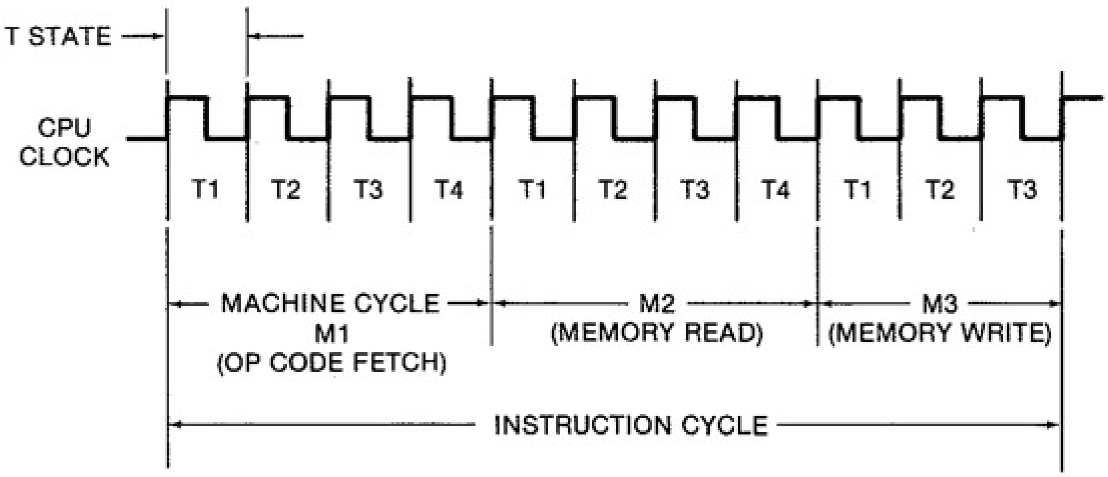


Fig. 4–4. CPU timing for executing a typical instruction. (*Courtesy MOSTEK Corp.*)

The instruction begins with an M1 cycle which is always an op-code fetch. During this cycle, the Z80 places the contents of the Program

Counter onto the address bus (during T1) and then performs a read of this location (during T2). The contents of this location (which is assumed to be the op code for a Z80 machine language instruction) is then sent to the instruction decoder circuit within the CPU. During the next two (or possibly more) T cycles, the CPU "decodes" the instruction op code. This is really nothing more than setting up certain flags, counters, and other circuits within the CPU to prepare it for the execution of the instruction. During the instruction decode T cycles, the Z80 places the contents of its internal Refresh counter onto the address bus and a "dummy" memory read is performed. No data is transferred during the read; its sole purpose is to allow for the refresh of dynamic RAMs. Since the refresh counter is incremented after every instruction, it will take 128 instructions to completely cycle through the entire 7-bit refresh counter values. At this point, however, every byte of dynamic RAM will have been refreshed. If the particular op-code read is for an instruction that requires more information (such as another op-code byte or an immediate mode operand) then another memory read will take place. This is accomplished by again putting out the PC address (which has automatically been incremented) and retrieving the next byte.

In the case of the INC (HL) instruction, the instruction decoder has informed the address control circuit that the contents of the H and L registers should next be placed onto the address bus (M2, T1 cycle). During the next T cycle, the contents of the memory location are read into the ALU. The instruction decoder has already set up the ALU to perform the "ADD 1" function. After the result is obtained, it is written into memory during the next machine cycle.

SPECIAL FUNCTIONS

There are four special functions that the Z80 can perform in response to hardware signals. These are the RESET, WAIT, INT, and NMI. Each of these functions is initiated when their corresponding pin on the Z80 chip is brought low.

RESET

A RESET signal causes the CPU to become initialized. This allows the system to begin operating from a well defined state. On the T/S

2068, the CPU is reset every time the computer is turned on. Initialization of the Z80 includes:

1. Loading the Program Counter with a value of 0000.
2. Disabling interrupts.
3. Loading the I and R registers with a value of 00.
4. Setting Interrupt Mode 0.

WAIT

When the WAIT line on the Z80 is brought low, the CPU will temporarily halt execution of its program. It will enter a "wait state" until this line is brought high again. The main purpose for this function is to allow the Z80 to communicate with devices that cannot read or write data as fast as the CPU. For example, when the CPU wants to read from RAM, it sets up the correct location on the address bus and then reads the data bus a short time later. It *assumes* that the memory device being read is fast enough to present its data within the specified amount of time after being addressed. If this is not the case, then the memory device can pull the WAIT line down for one or more clock cycles until its data is ready to be read by the CPU.

INT (Interrupt Request)

The Interrupt Request line is used to trigger an interrupt to the CPU. There are three different ways in which the Z80 can respond to such a signal. In the T/S 2068, an INT interrupt will cause the CPU to suspend its current operation and begin executing the program starting at location 38h. It is possible to enable and disable this function from software.

NMI (Nonmaskable Interrupt)

Like the INT interrupt, this signal will cause the CPU to stop executing its current program temporarily and begin running an interrupt service routine. For an NMI, the CPU performs a call to location 66h. This interrupt cannot be disabled from software.

ADDRESSING MODES

Most of the Z80 instructions operate on data stored in internal CPU registers, external memory, or in the I/O ports. Addressing refers to how the location of this data is generated in each instruction. There are ten different addressing modes possible with the Z80, although not every mode is available with each instruction. Here is a brief summary of each addressing mode.

Immediate Addressing

In this mode of addressing, the byte following the op code contains the actual operand.

op code	1 or 2 bytes
operand	

d7 dØ

An example of this type of instruction would be to load the accumulator with a constant where the constant is the byte immediately following the op code.

Immediate Extended Addressing

This mode is merely an extension of the immediate addressing mode in that the next two bytes following the op code are used as the operand.

op code	1 or 2 bytes
operand L	low-order byte
operand H	high-order byte

An example of this type of instruction would be to load the 16-bit HL register pair with two bytes of data.

Modified Zero Page Addressing

This mode applies to the special single-byte call instructions known as *restarts*. These instructions set the Program Counter to one of eight specific locations in page zero of memory. The benefit of this addressing mode is that it allows a single byte instruction to specify a complete 16-bit address. This saves memory space and shortens execution time.

Relative Addressing

When a program needs to execute instructions in a nonsequential order, the *jump* instruction is used. If the new address where the program is to continue executing is fairly close (less than 127 bytes away), then this location can be specified by a one byte offset from the program's current location. Using relative addressing, the byte following the op code is treated as a signed, twos-complement integer which is added to the address of the op code of the next instruction.

op code	Jump relative
operand	8-bit twos-complement displacement

There are two major advantages of using relative addressing. First, it allows the 16-bit jump address to be specified by a single byte for reduced memory usage and faster operation. Second, the use of relative addressing allows for *relocatable code*. This means that a program can be moved to any absolute location and still operate correctly.

Extended Addressing

Extended addressing means that the two bytes representing the address are included within the instruction.

op code	1 or 2 bytes
operand L	low-order address
operand H	high-order address

With extended addressing, a jump instruction can reach locations which are more than 127 bytes away.

Indexed Addressing

In this type of addressing, the byte following the op code contains a displacement which is added to one of the index registers (the op code specifies either IX or IY) to form the address of a memory location. The contents of the index register is not altered by this operation.

op code	2-byte op code
op code	
Displacement	8-bit twos-complement displacement added to index register to form memory address

Indexed addressing greatly simplifies programs that use tables, since the index register can point to the start of the table and the displacement can specify a given element within the table. Indexed addressing also facilitates the generation of relocatable code.

Register Addressing

Many of the Z80 op codes contain special bits which specify one of the CPU registers as the operand. This is known as register addressing. For example, we could have an instruction that would load the contents of register B into register C.

Implied Addressing

Implied addressing refers to operations where the op code automatically implies one or more of the CPU registers to be used as an operand. Instructions involving the I and R registers are examples of implied addressing.

Register Indirect Addressing

Register Indirect Addressing allows one of the 16-bit register pairs in the CPU to be used as a pointer to any location in memory. Since the address for the operand must already be loaded into the selected register pair, the instruction takes the form of a simple one- or two-byte op code.

An example of this type of instruction would be to load the accumulator with the contents of the memory location pointed to by the HL register. Such an instruction would be written in assembly language mnemonics as LD A, (HL). The use of parentheses around the HL operand signifies that this register is to be used as a pointer to a memory location. That is, when we write HL, it means we are interested in the contents of the *register pair* HL. But (HL) means that we want the contents of the memory location *pointed to* by the HL register.

Register Indirect Addressing can also be used to specify 16-bit operands. In this case, the contents of the register point to the first (lower) byte of the 16-bit operand. The register contents are then automatically incremented to obtain the address of the high order byte.

Bit Addressing

The Z80 has the ability to set, reset, or test any bit within a CPU register or memory location. Only the specified bit is affected (or examined). The actual location tested can be specified by either register, register indirect, or indexed addressing and the specific bit is indicated by three bits of the op code.

5
Memory Map of the T/S 2068

Since the Z80 has 16 address lines, it can address up to 65,535 different locations. Only six or seven years ago, 64K of memory was considered enormous. It represented the upper limit for personal computers. Today's programs have just begun to stretch this limit. The IBM PC uses a 16-Bit CPU capable of addressing *megabytes* of RAM. Eight-bit machines, such as the T/S 2068, however, must turn to a design called "bank switching" to increase accessible memory. Bank switching allows more than 64K of memory to be connected to a single 8-bit CPU. For this to work, two or more memory devices must be assigned the same physical memory address. The trick, therefore, is in keeping one, *and only one,* device active at any given time. This is the job of the T/S 2068's memory bank switching hardware.

While switching complete banks of 64K memory addresses is sometimes useful, in the T/S 2068 it proves more desirable to break each bank into several "chunks." Making each chunk 8K bytes long, divides the Z80 address range into 8 equal parts. These are labelled chunk 0 through chunk 7. Since the T/S 2068 is capable of controlling 256 different banks of memory, a memory map of the computer would look like Fig. 5–1. The banks are numbered from 0 to 255. Of course, more of these banks will be empty (i.e., they will not contain any RAM or ROM) so it would not be of much use to turn them on.

BANK SELECTION

There will always be eight and only eight chunks active at any point in time. Each chunk will have a different number and can reside in any of the 256 banks. Three of these banks are reserved for special purposes; the remaining 253 are called expansion banks and may be used by peripherals that connect to the back of the T/S 2068.

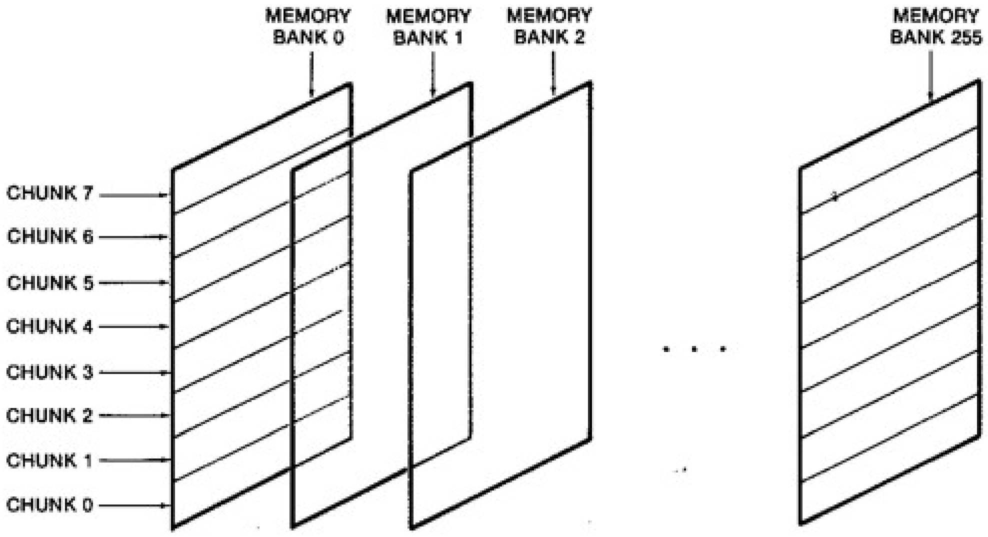


Fig. 5–1. Memory map of T/S 2068.

Home Bank

Bank number 255 is called the Home Bank. It contains most of the T/S 2068's built-in ROM and all of its RAM. It is the bank that gets selected by default and thus occupies the entire 64K space when the computer is first turned on. The Home ROM contains the BASIC interpreter, the fundamental I/O routines (display, keyboard, printer, etc.), and basic links to other peripheral devices through a channelled I/O section. To enable any chunks in the other banks, a special sequence of instructions must be executed by the CPU. We'll explain this technique shortly.

EXROM Bank

Bank number 254 is called the Extension ROM, or EXROM, bank. It contains only one ROM which occupies chunk #0. When selected, it replaces the first chunk of ROM in the Home Bank. Contained within the EXROM are the cassette tape I/O routines, bank switching code, and system initialization procedures. Initialization of the T/S 2068 can be quite complex and most of the machine code to do this resides in the EXROM. Of course, when the computer is first turned on, it is the program in the Home ROM that starts executing. Therefore one of the first things that the Home ROM does is to enable the EXROM and then jump to its initialization routine.

Dock Bank

Bank number 0 is called the Dock bank. It represents any memory contained within a Timex Command Cartridge. These cartridges are inserted into the compartment on the right side of the T/S 2068 which attaches them to the computer's dock connector. The Dock bank is usually occupied with some form of ROM Oriented Software.

Expansion Banks

The remaining 253 banks are available for use by peripherals that connect to the T/S 2068's rear card edge. To be compatible with the bank switching scheme employed by the T/S 2068, such peripherals must follow the hardware and software protocol set by Timex Computer. This will be described next.

BANK SWITCHING HARDWARE/SOFTWARE

All of this talk about bank switching sounds great but there are some restrictions. First of all, we need some sort of hardware to physically enable and disable each bank that we have. This hardware is built into the T/S 2068 for controlling the Home, EXROM, and Dock banks. A very elaborate bank switching-control scheme is also implemented in the T/S 2068 for handling the Expansion Banks.

Another problem associated with bank switched memory is that when the switch takes place, it must not affect the program flow of the routine that does the bank switching. That is, you would not want the CPU executing a program in one bank, and then suddenly find itself in the middle of some other routine in another bank. There are two ways around this problem. One way involves duplicating the bank switching code into the same memory locations of each bank that is to be switched. Therefore, as the bank switching routine executes and performs the switch, the CPU continues to execute the same program (although it is now in another bank).

The other way to solve this problem is to have one section of memory which is never switched out. This area can then contain the bank switching code as well as any other common information (such as interrupt handlers, machine stack, etc.). In the T/S 2068, memory chunk 3 is normally reserved for such use.

Bank Switching Control

The bank switching hardware within the T/S 2068 is accessed through four I/O ports:

DKHSPT	= F4h	Dock horizontal select port
BDATPT	= FCh	Expansion bank data port
BCMDPT	= FDh	Expansion bank address port
HREXPT	= FFh	Home ROM extension select port (bit7)

The Home Bank has the lowest priority, thus its chunks are enabled by default when no other Bank has the same chunks enabled. The DOCK Bank has the next highest priority, thus its chunks are enabled when no Expansion Bank has the same chunks enabled. The Expansion Banks have the highest priority. The Home ROM Extension Bank (EXROM) has the same priority as the DOCK Bank.

The Dock horizontal select port controls which of the DOCK bank chunks are to be enabled. By sending a given value out through this port, those chunk numbers corresponding to the '1' bits in this value will be enabled. Thus to activate only chunk number 7 in the DOCK Bank, we would OUTput the binary number 10000000 through port F4h. It is also possible to INput from this port and read back the current status (i.e., which DOCK chunks are enabled).

The EXROM is selected by writing a '1' to bit 7 of the HREXPT port (FFh). If this bit is set, the EXROM will overlay chunk 0 of the DOCK Bank. Thus to access this ROM, you must set bit 7 in port FFh *and* set at least bit 0 in the DKHSPT port (F4h). You must also insure that no external Banks have done their chunk 0 selected, since external Banks have higher priority than the DOCK Bank and, also, therefore the EXROM Bank.

HOLD	Temporary holding register
ABN	Assigned bank number (one for each expansion bank)
BNA	Bank number accessed register
HS	Expansion bank horizontal select register (one for each expansion bank)
STATUS	Status nybble* whose bits have the following interpretation:

bit 0—Set to Ø if bank caused an IRQ interrupt
bit 1—Not used
bit 2—Set to Ø if bank is responding to memory read/write
bit 3—Not used

*A nybble equals half of a byte or 4 bits.

Selecting an Expansion Bank is a little more complicated because we must use the HOLD register to latch data into and out of the BNA, ABN, and HS registers. The BNA register contains the number of the Bank whose status is being read or changed. The ABN register contains the Bank number assigned to a particular Bank. And the HS register specifies which chunks in that expansion Bank are enabled. Note that there is only one HOLD register and one BNA register. There are ABN and HS registers for each Expansion Bank, however.

There is no way to directly read or write to these registers from the Z80. The only way to access these registers is through the Expansion Bank Controller Registers shown in Table 5–1. These, in turn, are reached via the BDATPT (FCh) and BCMDPT (FDh) ports. The BCMDPT port is first used to address one of the Expansion Bank Controller Registers by sending out a 2-bit value. This selects one of the four controller registers which can then be read or written to by the BDATPT port. By writing to controller registers 0 or 1, we can perform one of the eight commands as listed in Table 5–2.

Table 5–1. Expansion Bank Controller Registers (Courtesy Timex Computer Corp.)

Address	Read Data Port	Write Data Port
Ø	Read status	Write command Type I
1	None	Write command Type II
2	Read HS ls nybble	Write hold reg. ls nybble
3	Read HS ms nybble	Write hold reg. ms nybble

If all this seems rather complicated, it's because it really is! An example will make things clearer. Suppose we have a device plugged onto the back of our T/S 2068 and it has some ROM which we would like to activate into chunk 0. We'll assume it is configured as bank number 1. We thus need to enable chunk 0 in Bank 1. Since Expansion Banks have the highest priority, we do not have to worry about turning off the Home ROM chunk 0, DOCK Bank chunk 0, or the EXROM. We would first, however, have to disable any other Expansion Banks that had chunk 0 active. Now we must load a binary 00000001 into the Horizontal Select (HS) register for bank number 1. To do this, we start

Table 5–2. Expansion Bank Controller Commands (Courtesy Timex Computer Corp.)

Type I Commands	
Value	**Function**
14	Reset controller — prepare to initialize.
13	Start interrupt REG sequence.
11	Initialization done. Move to next bank in daisy chain.
7	Reset interrupt flag.
Type II Commands	
Value	**Function**
14	Dump HOLD register to ABN.
13	Dump HOLD register to BNA.
12	Dump HOLD register to HS.
7	Not used.

by setting the Bank Number Access Register to 1. This is accomplished by writing 01 into the HOLD register (it takes two steps) and then causing the "Dump HOLD register to BNA" command to be executed. In machine language, this would look like:

```
OUT FD,2   Select control register 2
OUT FC,1   Write lower half of bank number into HOLD register
OUT FD,3   Select control register 3
OUT FC,0   Write upper half of bank number
OUT FD,1   Select control register 1
OUT FC,13  Send Type II command number 1 (Dump HOLD register to BNA)
```

Now that we have set the BNA register to point to the desired bank, we must set the corresponding HS register. In this example, our HS value is the same as the BNA value so the HOLD register is already loaded with the proper value. Thus to complete our task, we only need to perform:

```
OUT FC,11   Send Type II command number 2 (Dump HOLD register to HS)
```

With that, we have accomplished our goal. All of this assumes, however, that we have previously initialized the bank controller and that our Expansion Bank had been assigned a bank number of 1. Also note that it is possible to read a status byte associated with the bank. This byte is used to indicate whether the bank contains any RAM and also to indicate if the bank has generated an interrupt signal to the CPU.

HOME BANK MEMORY MAP

Fig. 5–2 shows the layout of the Home Bank on the T/S 2068. The labels along the outside of the map refer to system variables as described on pages 261-265 of the *T/S 2068 User Manual.* Some of these change as the computer is used; their initial values are shown in parentheses. When two display files are in use (e.g., in the 64-column mode), the machine stack and RAM-resident code are moved up to the top of RAM. The second display file then takes their place starting at location 6000h.

INPUT/OUTPUT FACILITIES

Nineteen of the Z80 ports have been assigned by the T/S 2068 as outlined in Table 5–3. Port FEh is used by a number of devices in the T/S 2068. When *reading* port FEh, the five least significant bits (D0-D4) represent the outputs from the keyboard and D6 corresponds to the Cassette Tape Input signal. By *writing* to Port FEh, we can set the border color (in D0, D1, and D2), set the tape output signal (D3), or toggle the internal speaker (D4).

Port FFh also controls a number of circuits within the T/S 2068 as follows:

Bit 0	— Enables D_FILE_2 (secondary display file)
Bit 1	— Enables ultra-high-color resolution mode
Bit 2	— Enables 64-column display (requires bit Ø to be set)
Bit 3	— Set paper color for 64-column display
Bit 4	
Bit 5	
Bit 6	— Disables keyboard interrupts
Bit 7	— Enables extension ROM in the EXROM bank

Table 5–3. T/S 2068 I/O Port Assignments (Courtesy Timex Computer Corp.)

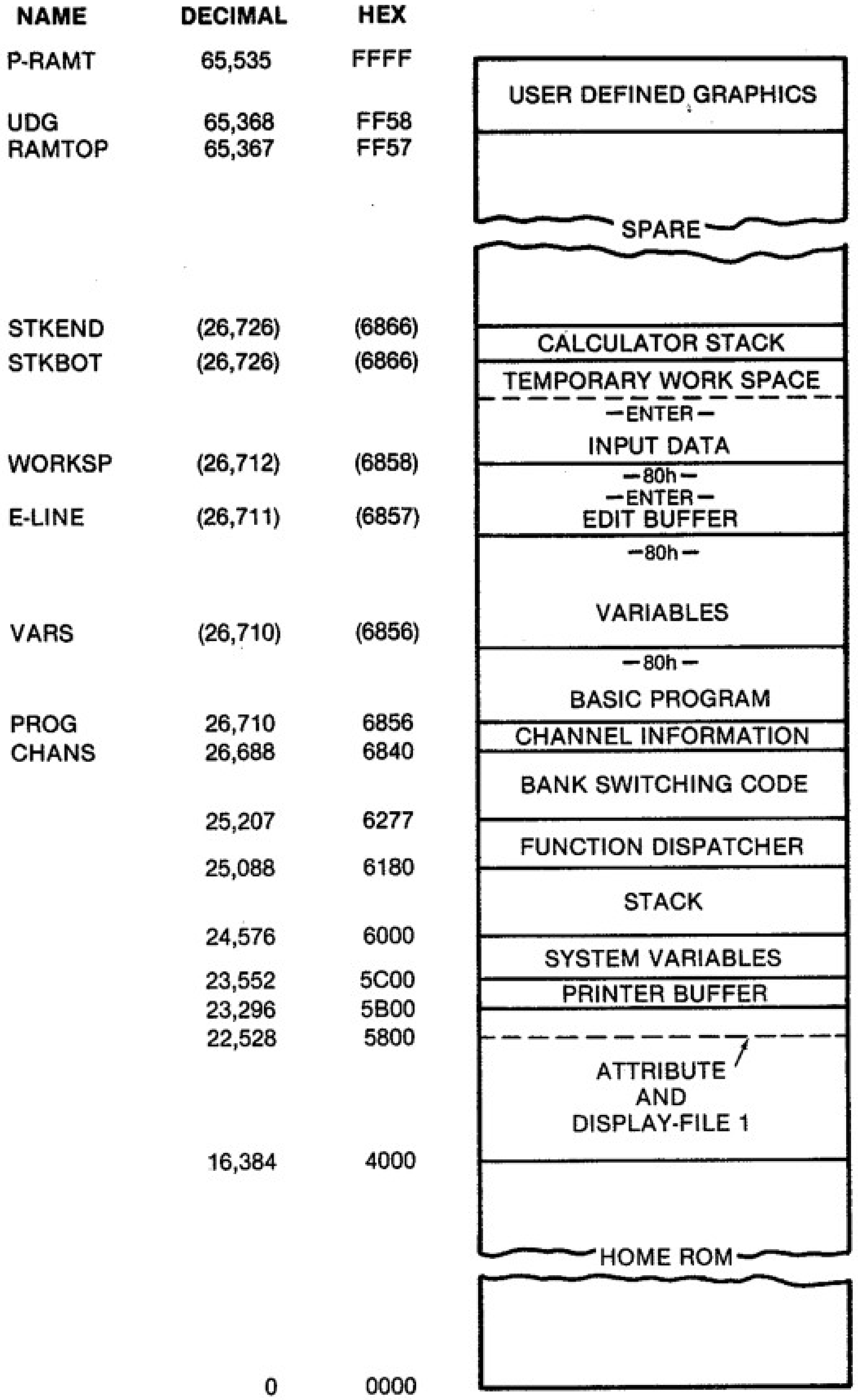
Port	Function
FF	Display Modes, Extension ROM, Interrupt Control
FE	Keyboard, Border Color, Tape, Beep
FD	Bank Controller — Address
FC	Bank Controller — Data
FB	Printer
FA	Printer
F9	Printer
F8	Printer
F7	Reserved
F6	PSG — Data
F5	PSG — Address
F4	DOCK Horizontal Select
F3	Printer
F2	Printer
F1	Printer
FØ	Printer
EF	Reserved

Notes: Modem uses ports C7, CF, D7, DF.
Timex devices will use ports 78-FFh.
Other vendors will use ports 00-77h.

KEYBOARD

The keyboard on the T/S 2068 is divided into a matrix of eight "half rows" with five keys each. When a key is pressed, it makes a connec-

HOME BANK MEMORY MAP



NAME	DECIMAL	HEX
P-RAMT	65,535	FFFF
UDG	65,368	FF58
RAMTOP	65,367	FF57
STKEND	(26,726)	(6866)
STKBOT	(26,726)	(6866)
WORKSP	(26,712)	(6858)
E-LINE	(26,711)	(6857)
VARS	(26,710)	(6856)
PROG	26,710	6856
CHANS	26,688	6840
	25,207	6277
	25,088	6180
	24,576	6000
	23,552	5C00
	23,296	5B00
	22,528	5800
	16,384	4000
	0	0000

Fig. 5–2. Home Bank memory map (Numbers in parenthesis represent initial values for system variables — these will change as computer is used.)

tion between its "row line" and "column line." To detect when a key has been pressed, the computer repetitively scans through all eight rows, looking for a signal that represents a keypress. On the T/S 2068, this is accomplished by connecting the eight row lines to the upper half of the address bus. The five column lines connect to the CPU through I/O port FEh.

By sequentially lowering each of the address lines A8–A15 and then reading the I/O port, the CPU can tell what key, if any, has been pressed. Normally, with no keys pressed, the five keyboard bits read through the port will all be high no matter what address is used. When a key is pressed, however, it will place a low on its respective I/O bit whenever its row address line is also low. Therefore, by detecting the column bit and knowing the row position, the keyboard polling routine can determine which key was pressed. Table 5–4 describes how each key is mapped. D0 represents the key closest to the outside of each row; D4 the one nearest the middle of the keyboard. A bit is 0, if key is pressed; 1 if it is not.

SOUND

There are two ways to generate sound on the T/S 2068. By toggling bit 4 of port FEh, many types of sounds can be generated under software control. This usually must be done in machine language and it may require all of the CPU's time just to generate the sound. To see how this works, type in the following one-line program:

```
10 OUT 254,7: OUT 254,23: GO TO 10
```

When this is RUN, the computer will generate a low pitched tone. Notice that this is about the fastest BASIC program that can toggle the port as required. Therefore, this program creates the highest pitched tone possible from BASIC. Of course, you can add other statements to slow this program down if you want lower pitched notes. With machine language, however, we can create tones that are so high in frequency that they cannot even be heard.

In case you haven't figured out what this BASIC program does, here is a quick rundown. The OUT command is used to send data to the port number 254 (FEh). The first statement puts out a value of seven. This sets the speaker toggle (bit 4) to a zero and leaves the border color white (by setting bits 0, 1, and 2 high). The next statement comes along and outputs a value of 23. This sets the speaker bit high while

Table 5–4. Hardware Map of T/S 2068 Keyboard

Port Address		Key Indicated When Bit Is Low				
Decimal	Hex	Bit4	Bit3	Bit2	Bit1	BitØ
32766	7FFE	B	N	M	SYMBL	SPACE*
4915Ø	BFFE	H	J	K	L	ENTER
57342	DFFE	Y	U	I	O	P
61438	EFFE	6	7	8	9	Ø
63486	F7FE	5	4	3	2	1
6451Ø	FBFE	T	R	E	W	Q
65Ø22	FDFE	G	F	D	S	A
65278	FEFE	V	C	X	Z	CAPS

*Same as BREAK key.

keeping the border color bits the same. Thus the speaker will toggle one time, putting out a small click. When the final statement is executed, the program returns to the beginning of the line and starts over. While the BASIC interpreter is relatively slow for this type of task, it is still able to execute this line about 200 times per second, creating the low-pitched tone.

The T/S 2068 also contains a Programmable Sound Generator chip which is tied to the CPU through ports F5h and F6h. The first port is used to address one of the 15 registers in the PSG. The other port is then used to read or write data to these registers. See Chapter 7 for more details on the PSG. The sounds generated by these two methods are combined to drive the speaker underneath the T/S 2068. The software generated sounds also feed the cassette MIC jack. Therefore, these sounds can be recorded on the cassette machine or fed into an amplifier, if desired. The sounds generated by the PSG do not come out this jack but there is a signal available on the rear card edge connector.

JOYSTICKS

The two joystick connectors are read through the I/O port on the PSG. To access this port, a value of 14 must be written to the PSG address port (F5h). Bit 8 of the address bus is used to select one of the two joystick connectors. The CPU can then determine the status of the selected joystick by reading the PSG data port (F6h). The byte read is interpreted as follows:

Bit 0	— 0 indicates stick is UP
Bit 1	— 0 indicates stick is DOWN
Bit 2	— 0 indicates stick is LEFT
Bit 3	— 0 indicates stick is RIGHT
Bit 4	
Bit 5 }	— Not used (all ones)
Bit 6	
Bit 7	—0 indicates button is depressed

6
Connecting to the Outside World

If "No man is an island," then no computer is a complete data processing system. To do anything useful, we need to attach a variety of peripherals such as a television or monitor, tape recorder, printer, modem, etc. All of these devices connect to the T/S 2068 through a variety of special connectors.

By now, you are probably familiar with the POWER, TV, MONITOR, EAR, and MIC connectors. The power-supply unit that plugs into the wall supplies electricity to operate the T/S 2068 through the POWER connector. To create a visual display on your television set or video monitor, the computer generates a special electronic waveform called a *composite video* signal. (It is also possible to connect a monitor directly to the computer's RGB signals — this will be discussed in the next chapter.) There are three places on the back of the T/S 2068 where this composite video signal is available. The first place is at the MONITOR jack which is the normal place to connect a video monitor. This same signal is also available on one of the pins of the peripheral expansion connector. The video signal also goes to an RF modulator within the T/S 2068 where it is converted into a radio frequency signal much like the one broadcast from your local tv station. This allows the computer to be conveniently connected to the antenna terminal of a tv set, usually through a transfer switch box. The computer's video signal is then seen by tuning the tv to the proper channel.

For program and data storage, most T/S 2068 owners use a simple cassette recorder. The computer contains all of the hardware and software needed to support such a mass storage device. Two connectors are used to hook up a recorder. The MIC jack sends out a very small signal (on the order of the level produced by a microphone — about 150 mV). The signal can then be fed into the MIC jack of the recorder and used to store data from the computer on tape. See Chapter 15 for details on this signal. When the tape is played back, the signal from the recorder's EAR phone output is then fed into the EAR jack on the T/S 2068. The computer can "listen" to this signal and then reconstruct the data that originally was in the machine when the tape was SAVE'd.

The output level from a cassette recorder can vary according to the setting of its volume control. For proper operation, the T/S 2068 requires a signal level of several volts. By the way, any software generated sounds such as the "keyclick" or BEEP commands, also come out the MIC jack.

JOYSTICK CONNECTORS

There are two joystick connectors on the T/S 2068 — one on each side of the machine. These connectors are designed to accept standard eight-position joysticks with 9-pin "D" type plugs. These are compatible with joysticks used on the Atari and Commodore computers, as well as most video games. Fig. 6–1 and Table 6–1 show the pin assignments for these connectors.

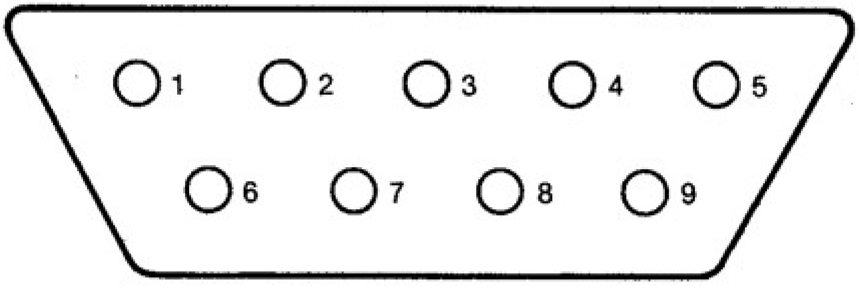


Fig. 6–1. Joystick connectors.

When the "stick" is moved away from its center position, it will close one (or two) switches within the joystick. These switches will then connect the Read Strobe line to the corresponding direction output pin. When the joystick port is being accessed, the Read Strobe signal will go low. Thus, any direction pins whose switch is closed will also go low. The other direction outputs will remain high. By reading the port data, the computer can determine which, if any, joystick switches are closed.

If you are not going to use joysticks, these connectors can be used for other general-purpose I/O. The direction pins, for example, can be connected to any "TTL-compatible" circuit. This would give up to ten input lines that could be used to check the status of various external

Table 6–1. Joystick Connector Pin Assignments

Pin	Name	Description
1	$\overline{\text{DIR1}}$	Direction One (UP)
2	$\overline{\text{DIR2}}$	Direction Two (DOWN)
3	$\overline{\text{DIR3}}$	Direction Three (LEFT)
4	$\overline{\text{DIR4}}$	Direction Four (RIGHT)
5	—	Not used
6	$\overline{\text{BUTTON}}$	Button Input
7	—	Not used
8	$\overline{\text{RDSTB}}$	Read Strobe. This pin goes low for approx. 3 μS when the joystick port is being read. It is an open-collector output.
9	—	Not used

devices. By using the STICK command, these inputs can easily be read by a BASIC program. The Read Strobe output signal can also be used to trigger an external device. This pin goes low when active, but requires a pull-up resistor to make it go high when inactive. Adding a 10K-ohm resistor between Read Strobe and +5 Volts should work just fine. Since the +5-volt line is not available at this connector, an alternative is just to tie the $\overline{\text{RDSTB}}$ signal to one of the direction input lines. This will give it the necessary pull up (via a resistor on the direction input line) but makes that direction input unusable. Note that the $\overline{\text{RDSTB}}$ signal is a single, very narrow pulse. It stays low for only about 3 microseconds (that's 3 millionth's of a second!) and then goes high again. You can also generate a $\overline{\text{RDSTB}}$ signal from BASIC using the STICK command.

DOCK CONNECTOR

Inside the Timex Command Cartridge slot there is a 36-pin edge connector. This allows any ROM Oriented Software cartridges placed

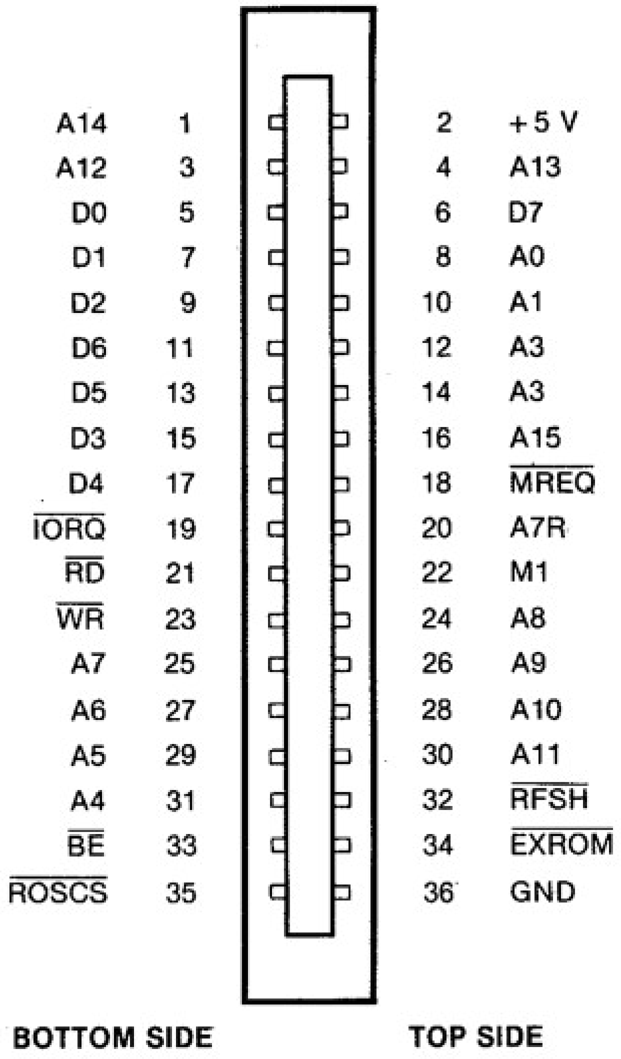
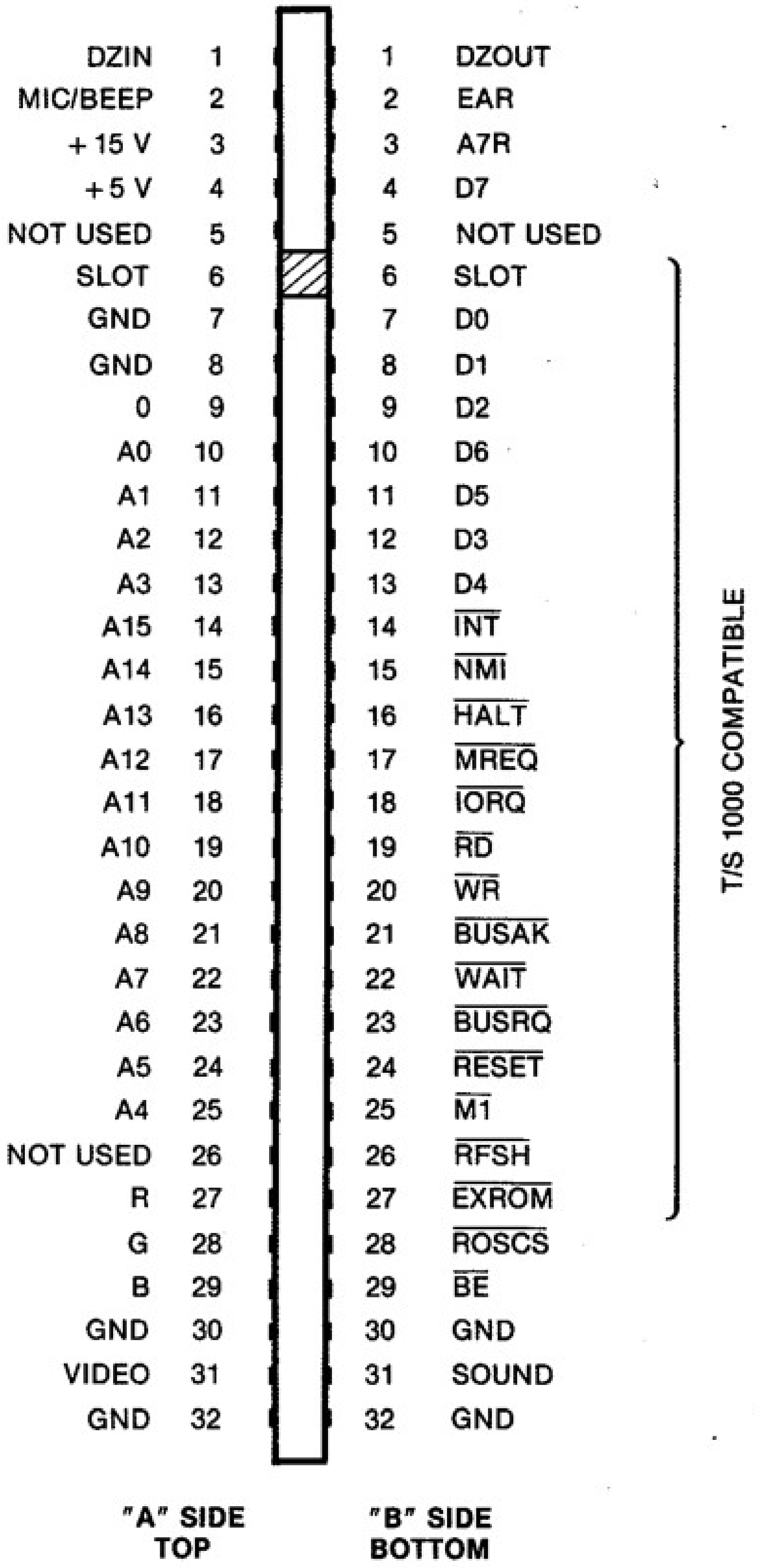


Fig. 6–2. DOCK connector.

in this connector to interface with the computer. Fig. 6–2 and Table 6–2 show the layout and function of each pin on this connector. Since the DOCK port is used primarily to support ROM software, the signals available at this connector are related to the data bus, address bus, and bank switching hardware.

PERIPHERAL EXPANSION CONNECTOR

On the back of the T/S 2068 behind a small cover, there is a 64-pin card edge for peripheral expansion. This is where the T/S 2040 printer and T/S 2050 modem are attached. The layout of this card edge is shown in Fig. 6–3 and a description of each signal is given in Table 6–3.



"A" SIDE TOP			"B" SIDE BOTTOM
DZIN	1	1	DZOUT
MIC/BEEP	2	2	EAR
+ 15 V	3	3	A7R
+5 V	4	4	D7
NOT USED	5	5	NOT USED
SLOT	6	6	SLOT
GND	7	7	D0
GND	8	8	D1
0	9	9	D2
A0	10	10	D6
A1	11	11	D5
A2	12	12	D3
A3	13	13	D4
A15	14	14	$\overline{\text{INT}}$
A14	15	15	$\overline{\text{NMI}}$
A13	16	16	$\overline{\text{HALT}}$
A12	17	17	$\overline{\text{MREQ}}$
A11	18	18	$\overline{\text{IORQ}}$
A10	19	19	$\overline{\text{RD}}$
A9	20	20	$\overline{\text{WR}}$
A8	21	21	$\overline{\text{BUSAK}}$
A7	22	22	$\overline{\text{WAIT}}$
A6	23	23	$\overline{\text{BUSRQ}}$
A5	24	24	$\overline{\text{RESET}}$
A4	25	25	$\overline{\text{M1}}$
NOT USED	26	26	$\overline{\text{RFSH}}$
R	27	27	$\overline{\text{EXROM}}$
G	28	28	$\overline{\text{ROSCS}}$
B	29	29	$\overline{\text{BE}}$
GND	30	30	GND
VIDEO	31	31	SOUND
GND	32	32	GND

Fig. 6–3. Expansion edge connector.

Most of the data and address lines are not internally buffered by the T/S 2068 and, therefore, must be used with care by any peripherals. If these lines must be connected to several other devices, they should first be buffered by a suitable circuit. Just about every useful signal within the T/S 2068 is available at the expansion connector, including all of those at the DOCK connector. A portion of the 64-pin expansion card edge is compatible with the 46-pin card edge used by the T/S 1000 computer. Those pins which are identical are shown in Fig. 6–3. This compatibility makes it possible to use some peripherals (such as the T/S 2040 printer) with either computer. In general, however, devices designed for the T/S 1000, T/S 1500, or the Sinclair Spectrum will not work with the T/S 2068 computer.

Table 6–2. DOCK Connector Pin Assignments

Pin	Name	Description
1,3,4 8,10,12 14,16 24-31	A0–A15	The 16-bit address bus. Only A13 and A14 have been buffered on the T/S 2068.
2	+5V	+5-volt power supply.
5-7,9 11,13 15,17	D0–D7	The 8-bit bidirectional data bus.
18	$\overline{\text{MREQ}}$	Memory Request. This line goes low when the Z80 is ready to access a memory location.
19	$\overline{\text{IORQ}}$	I/O Request. This line goes low when the Z80 is ready to access an I/O port.
20	A7R	Refresh address bit 7.
21	$\overline{\text{RD}}$	Read. This signal goes low when the Z80 wants to read data from a memory location or I/O device.
22	$\overline{\text{M1}}$	CPU M1 state. This signal indicates when the Z80 is performing an M1 (instruction fetch) cycle.

Table 6–2 — cont. DOCK Connector Pin Assignments

Pin	Name	Description
23	$\overline{\text{WR}}$	Write. This signal goes low when the Z80 wants to write to a memory location or I/O device.
32	$\overline{\text{RFSH}}$	Refresh. This line signals when the Z80 is putting out a refresh address on the lower half of the address bus.
33	$\overline{\text{BE}}$	Bank Enable. When this line is pulled low, it disables the RAM/ROM inside the T/S 2068. This allows the cartridge memory to be accessed conflicting with the internal memory.
34	$\overline{\text{EXROM}}$	Extension ROM enable. This signal is used to access the EXROM.
35	$\overline{\text{ROSCS}}$	Rom Oriented Software Chip Select. When this signal goes low it indicates that the CPU wishes to access the RAM or ROM on the ROS cartridge.
36	GND	System electrical ground.

Table 6–3. Expansion Edge Connector Pin Assignments

Pin	Name	Description
1A	DZOUT	Daisy chain out. This signal is used to control the selection of the Expansion memory banks.
2A	EAR	This line is a duplicate of the signal at the EAR jack on the rear of the computer. It can be used as either an input or output.

Table 6–3 — cont. Expansion Edge Connector Pin Assignments

Pin	Name	Description
3A	A7R	Refresh address bit 7.
4A, 7–13A	D0–D7	This is the 8-bit bidirectional data bus.
5A	—	Not used.
6A	—	Slot.
14A	$\overline{\text{INT}}$	Interrupt Request. When this line is pulled low, the T/S 2086 begins an interrupt cycle if the interrupt flag (IFF) has been set. Otherwise, this line is ignored.
15A	$\overline{\text{NMI}}$	Nonmaskable Interrupt. When this line is pulled low, the T/S 2068 will begin executing an NMI cycle. This causes the Z80 to perform a restart to location 0066h.
16A	$\overline{\text{HALT}}$	This signal indicates that the Z80 has executed a HALT instruction and is awaiting either an INT or NMI interrupt before operation can resume.
17A	$\overline{\text{MREQ}}$	Memory Request. This line goes low when the Z80 is ready to access a memory location.
18A	$\overline{\text{IORQ}}$	I/O Request. This line goes low when the Z80 is ready to access an I/O port.
19A	$\overline{\text{RD}}$	Read. This signal goes low when the Z80 wants to read data from a memory location or I/O device.
20A	$\overline{\text{WR}}$	Write. This signal goes low when the Z80 wants to write to a memory location or I/O device.

Table 6–3 — cont. Expansion Edge Connector Pin Assignments

Pin	Name	Description
1B	DZIN	Daisy in. This signal is used with the Daisy out line to coordinate access of the Expansion memory banks.
2B	MIC/ BEEP	This line carries the same signal as the MIC jack but at a higher level. It contains pulses generated by the cassette write and software BEEP routines.
3B	+15V	+15 volt unregulated. Actually this can be as high as +21 volts.
4B	+5V	+5-volt regulated power supply.
5B	—	Not used.
6B	—	Slot.
7B,8B, 30B,32B	GND	System electrical ground.
9B	O	CPU system clock (inverted).
10–25B	A0–A15	This is the 16-bit address bus.
26B	—	Not used.
27B	R	Red color signal. (TTL level — positive).
28B	G	Green color signal. (TTL level — positive).
29B	B	Blue color signal. (TTL level — positive).
31B	VIDEO	Composite video signal. This is a duplicate of the signal available at the MONITOR connector. Approximately 1V peak-to-peak.
21A	$\overline{\text{BUSAK}}$	Bus Acknowledge. This line is used to indicate that the CPU address, data, and control signals have been deactivated so that an external device can now control these signals.

Table 6–3 — cont. Expansion Edge Connector Pin Assignments

Pin	Name	Description
22A	$\overline{\text{WAIT}}$	Pulling this line low will cause the CPU to enter wait states for as long as the line is held low.
23A	$\overline{\text{BUSRQ}}$	Bus Request. This line is used to signal the Z80 that an external device wishes to take control of the data and address buses.
24A	$\overline{\text{RESET}}$	Pulling this line low causes the computer to perform a reset operation as if it were just turned on. This line can also be used as an output to reset external hardware whenever the computer is reset.
25A	$\overline{\text{M1}}$	CPU M1 state. This signal indicates when the Z80 is performing an M1 (instruction fetch) cycle.
26A	$\overline{\text{RFSH}}$	Refresh. This line signals when the Z80 is putting out a refresh address on the lower half of the address bus.
27A	$\overline{\text{EXROM}}$	Extension ROM enable. This signal is used to access the EXROM.
28A	$\overline{\text{ROSCS}}$	Rom Oriented Software Chip Select. When this signal goes low it indicates that the CPU is accessing the RAM or ROM on a cartridge in the DOCK connector.
29A	$\overline{\text{BE}}$	Bank Enable. When this line is pulled low, it disables the RAM/ROM inside the T/S 2068. This allows the Z80 to communicate with an External memory bank without conflicting with the internal memory.
30A,32A	GND	System electrical ground.
31A	SOUND	This is the analog output from the PSG.

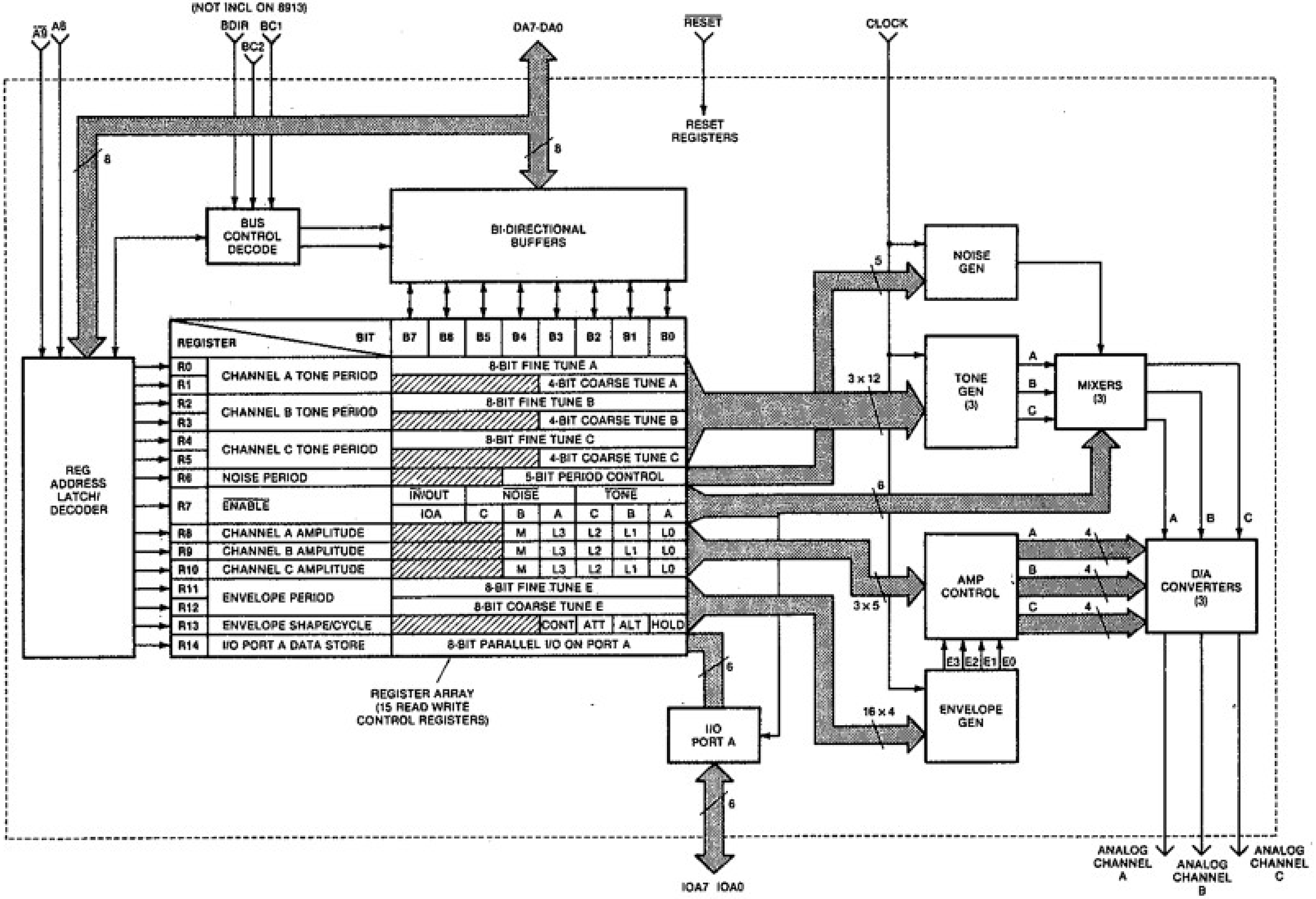
7
Using the T/S 2068's Programmable Sound Generator

The T/S 2068 has a built-in, sound-generating IC. This sophisticated "chip" is known as the Programmable Sound Generator, or PSG for short. (Technically speaking, the PSG is a General Instrument AY-3-8912.) Telling the PSG what to do is simply a matter of sending the proper bytes of data out through a pair of I/O ports. Specifically, the T/S 2068 uses ports F5h and F6h for communicating with the PSG. Port F5h is used to address one out of 15 registers within the PSG, and the eight bits of data are written to (or read back from) the PSG via port F6h. From BASIC, there is a SOUND command which makes this task slightly easier.

INSIDE THE PSG

If we could look inside the PSG, we would find 15 byte-wide registers arranged as shown in Fig. 7–1. These registers control the sound generated by this IC. To generate music or sound effects, the computer must send a series of commands to the PSG at precise times. In between commands, the PSG will continue to produce sound according to its last instructions. Thus, the CPU is free to perform other tasks, such as updating the screen or graphics display. This is one of the main advantages of using a hardware PSG versus the software generated sound.

The PSG can be broken down into six major functions (see Fig. 7–2). We start with three identical tone generators. Each generator produces a square-wave signal at a software-controlled frequency. The exact frequency is determined by programming the coarse tune and fine tune registers associated with each tone generator. In addition to the three tone oscillators, there also is a random-frequency



REGISTER		BIT B7	B6	B5	B4	B3	B2	B1	B0
R0	CHANNEL A TONE PERIOD	8-BIT FINE TUNE A							
R1		▨	▨	▨	▨	4-BIT COARSE TUNE A			
R2	CHANNEL B TONE PERIOD	8-BIT FINE TUNE B							
R3		▨	▨	▨	▨	4-BIT COARSE TUNE B			
R4	CHANNEL C TONE PERIOD	8-BIT FINE TUNE C							
R5		▨	▨	▨	▨	4-BIT COARSE TUNE C			
R6	NOISE PERIOD	▨	▨	▨	5-BIT PERIOD CONTROL				
R7	$\overline{\text{ENABLE}}$	$\overline{\text{IN}}$/OUT		$\overline{\text{NOISE}}$			$\overline{\text{TONE}}$		
		IOA		C	B	A	C	B	A
R8	CHANNEL A AMPLITUDE	▨	▨	▨	M	L3	L2	L1	L0
R9	CHANNEL B AMPLITUDE	▨	▨	▨	M	L3	L2	L1	L0
R10	CHANNEL C AMPLITUDE	▨	▨	▨	M	L3	L2	L1	L0
R11	ENVELOPE PERIOD	8-BIT FINE TUNE E							
R12		8-BIT COARSE TUNE E							
R13	ENVELOPE SHAPE/CYCLE	▨	▨	▨	▨	CONT	ATT	ALT	HOLD
R14	I/O PORT A DATA STORE	8-BIT PARALLEL I/O ON PORT A							

Fig. 7–1. Register layout of AY-3-8912 PSG. (*Courtesy General Instrument Corp.*)

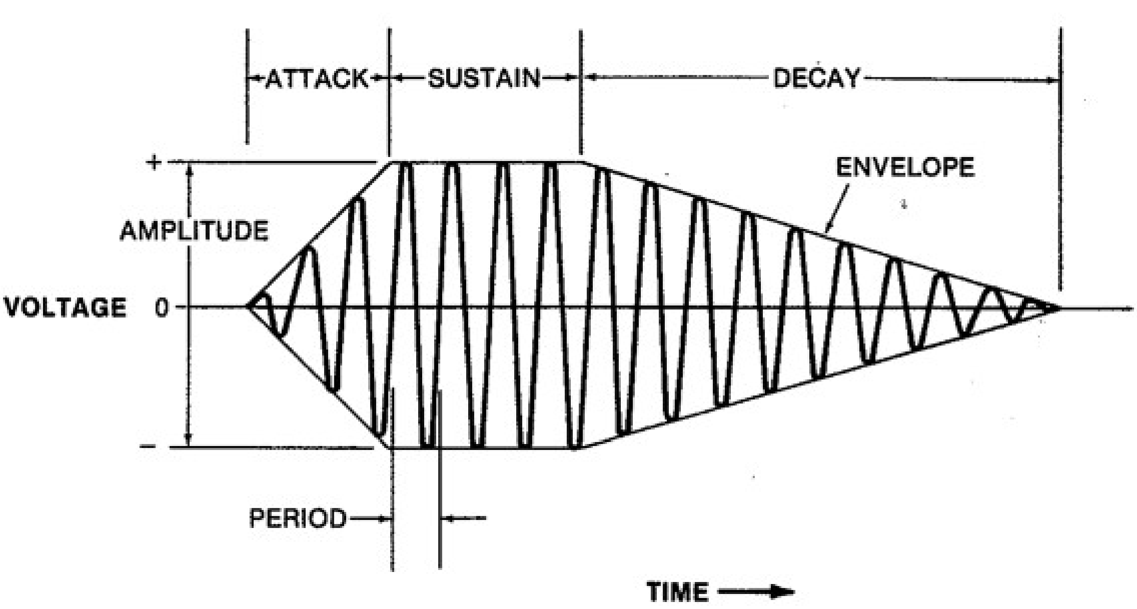


Fig. 7–2. Block diagram of PSG. (*Courtesy General Instrument Corp.*)

noise generator. This block creates a sound similar to the noise heard from a tv set when it is tuned to a channel that is not broadcasting. By controlling the sound produced by the noise generator, we can create the sound of percussion instruments, as well as many sound effects. The outputs from these tone generators and the noise generator are fed into a mixer section which controls which sound is to be fed into three D/A (digital to analog) converters. The purpose of these devices is to give control over the amplitude (loudness) of the signal.

The D/A converters set the volume for each channel according to the data stored in the amplitude control registers. These registers can supply a fixed 4-bit value representing one of 16 logarithmically scaled amplitudes. Alternatively, each register can specify that the amplitude be controlled by the envelope generator which automatically causes the amplitude to rise or fall at a predetermined rate. Before describing how the PSG is programmed, we'll discuss a few basics of sound.

AUDIO BASICS

Sound is nothing more than a compression wave — molecules of air moving closer together and then farther apart. This wave is usually generated by some vibrating object. In the case of the T/S 2068, it comes from the moving diaphragm, or cone, of a small speaker inside the computer. When this speaker is fed an alternating current, it will move in and out according to the varying potential (voltage) across its terminals. If this signal is made to alternate somewhere in the range of 20 to 20,000 times per second, then our ears can detect (hear) the motion of the speaker.

There are three characteristics of the voltage signal that can be discerned by our ears. First of all, there is the number of times that the voltage changes polarity (goes from positive to negative or vice versa). This determines the *frequency* or pitch of the signal. At 20 polarity reversals per second, the sound will be extremely low, almost like an earth tremor. To make the sound of a middle C note on the piano, the signal must have a frequency of about 260 cycles per second. The high end of the audio frequencies is at 20,000 cycles per second. This is usually written as 20,000 Hz or 20 KHz (Hz represents hertz, the unit of frequency and K represents 1000).

Sometimes it is useful to refer to the inverse function of frequency. That is, how many *seconds* (or fraction thereof) it takes to complete one cycle. This is known as the *period* of a signal and can be expressed in seconds, milliseconds (thousandths of a second), etc. Converting from frequency to period, or vice versa, is simply a matter of dividing the number by 1. For example, a 200-Hz signal has a period of 1/200 or .005 second (5 milliseconds).

Another characteristic of the voltage signal driving a speaker is the *amplitude* or level of the signal. This determines how *loud* the sound emanating from the speaker will be. The amplitude can be constant, as in the dial tone you hear when you pick up a telephone, or it can vary in some manner to create a different sound. When you hit a cymbal for instance, the sound starts off very loud. As the cymbal continues to vibrate, the sound begins to die off, getting softer and softer until there is no sound at all.

Sounds that have a varying amplitude can be broken up into three phases. The first phase consists of the time when the sound is beginning to increase in amplitude. This is known as the *attack* portion of the signal. If the signal maintains this amplitude for some length of time, then this is called the *sustain* portion. Finally, the sound enters the *decay* phase where the amplitude decreases, usually to zero. Sometimes repetitive combinations of these phases are strung together to form complex sounds. By drawing a line around the peak values of a given waveform, we can easily see what kind of variations are taking place in the signal's amplitude. This is known as the *envelope* of the signal (see Fig 7–3.)

<table>
<tr><th colspan="2">REGISTER BIT</th><th>B7</th><th>B6</th><th>B5</th><th>B4</th><th>B3</th><th>B2</th><th>B1</th><th>B0</th></tr>
<tr><td>R0</td><td rowspan="2">CHANNEL A TONE PERIOD</td><td colspan="8">8-BIT FINE TUNE A</td></tr>
<tr><td>R1</td><td colspan="4">▨</td><td colspan="4">4-BIT COARSE TUNE A</td></tr>
<tr><td>R2</td><td rowspan="2">CHANNEL B TONE PERIOD</td><td colspan="8">8-BIT FINE TUNE B</td></tr>
<tr><td>R3</td><td colspan="4">▨</td><td colspan="4">4-BIT COARSE TUNE B</td></tr>
<tr><td>R4</td><td rowspan="2">CHANNEL C TONE PERIOD</td><td colspan="8">8-BIT FINE TUNE C</td></tr>
<tr><td>R5</td><td colspan="4">▨</td><td colspan="4">4-BIT COARSE TUNE C</td></tr>
<tr><td>R6</td><td>NOISE PERIOD</td><td colspan="3">▨</td><td colspan="5">5-BIT PERIOD CONTROL</td></tr>
<tr><td rowspan="2">R7</td><td rowspan="2">$\overline{\text{ENABLE}}$</td><td colspan="2">$\overline{\text{IN}}$/OUT</td><td colspan="3">$\overline{\text{NOISE}}$</td><td colspan="3">$\overline{\text{TONE}}$</td></tr>
<tr><td colspan="2">IOA</td><td>C</td><td>B</td><td>A</td><td>C</td><td>B</td><td>A</td></tr>
<tr><td>R8</td><td>CHANNEL A AMPLITUDE</td><td colspan="3">▨</td><td>M</td><td>L3</td><td>L2</td><td>L1</td><td>L0</td></tr>
<tr><td>R9</td><td>CHANNEL B AMPLITUDE</td><td colspan="3">▨</td><td>M</td><td>L3</td><td>L2</td><td>L1</td><td>L0</td></tr>
<tr><td>R10</td><td>CHANNEL C AMPLITUDE</td><td colspan="3">▨</td><td>M</td><td>L3</td><td>L2</td><td>L1</td><td>L0</td></tr>
<tr><td>R11</td><td rowspan="2">ENVELOPE PERIOD</td><td colspan="8">8-BIT FINE TUNE E</td></tr>
<tr><td>R12</td><td colspan="8">8-BIT COARSE TUNE E</td></tr>
<tr><td>R13</td><td>ENVELOPE SHAPE/CYCLE</td><td colspan="4">▨</td><td>CONT</td><td>ATT</td><td>ALT</td><td>HOLD</td></tr>
<tr><td>R14</td><td>I/O PORT A DATA STORE</td><td colspan="8">8-BIT PARALLEL I/O ON PORT A</td></tr>
</table>

Fig. 7–3. Basic characteristics of an audio signal.

The last important characteristic of a signal is its *waveshape.* This determines the *timbre* or voice of the sound. It's what makes a middle C note on the piano sound different from the same note played on a guitar. Actually it is the presence of *harmonics,* or multiples of the primary frequency, that give each instrument its individual sound. The only pure sound is that created by a sine wave which consists of only one frequency without any harmonics. A tuning fork creates such monochromatic sound. Other waveshapes such as the "square wave" signals created by the T/S 2068 are rich in harmonics. When we speak of the frequency of such a signal, we always refer to the most dominant or *fundamental frequency.* While there may be numerous harmonics present in the signal, they are all considerably lower in amplitude than the fundamental.

PROGRAMMING THE PSG

To understand how the PSG creates sounds, let's start with the input clock and the register array. These two criteria fully define the actual sounds that come out of the PSG. First, we have a master clock signal which the T/S 2068 creates and feeds to the PSG's *CLOCK* input. This signal is nothing more than a square wave oscillating at 1.764 MHz (that's millions of hertz). Obviously this frequency is much too high to be heard by the human ear, so the PSG must reduce the signal into the audible range to be of any use. It does this with a series of circuits known as *dividers* since they divide the input frequency by some fixed amount to produce a lower frequency. The first thing that happens to the input clock is that it gets divided by 16. This, then, becomes the PSG's master clock which feeds the rest of the circuits.

Setting the Frequency

The PSG has three separate tone generators which can be individually programmed to create a single note. The actual frequency produced by each generator is equal to the PSG master clock divided by a 12-bit Tone Period value stored in the PSG's registers. For example, the Tone Period for the "A" channel of the PSG is comprised of Register 0 and part of Register 1. Fig. 7–4 shows a graphic representation of how this 12-bit value is formed. Although one half of Register 1 (hereafter referred to as R1) is not used, the lower four bits represent the most significant portion of the tone value. Therefore, R1 is called the Coarse Tune Register and R0 is the Fine Tune Register. Likewise for the other two channels B and C, there are two pairs of tuning registers, R2/R3 and R4/R5, respectively.

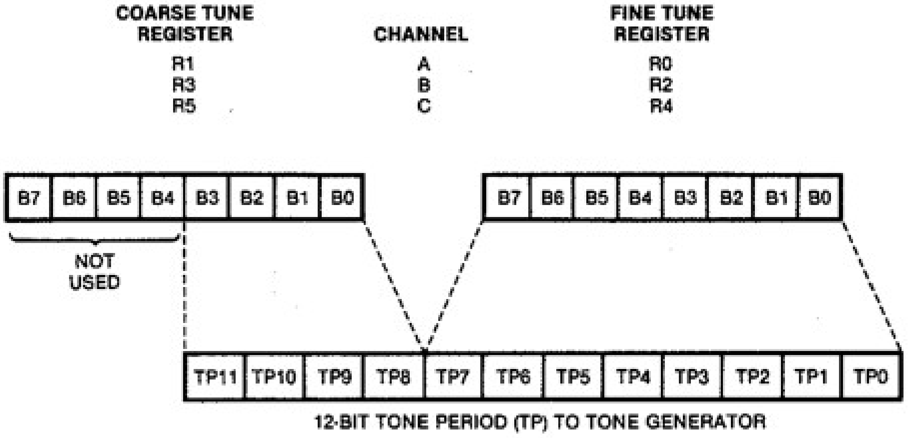


Fig. 7–4. Tone generator timing registers. (*Courtesy General Instrument Corp.*)

Let's see what kind of tones we can generate with the PSG. With a 12-bit divisor, we can represent any number between 1 and 4095 (dividing by 0 is not allowed). With a 1.764 MHz input clock divided by 16, we have a PSG master clock frequency of 110.25 KHz. This is the highest frequency obtainable using a tone period divisor of 1. At the low end, we can get 110.25 KHz divided by 4095, or 27 Hz. This covers most of the useful audio spectrum.

Note that the numbers stored in the Tune registers represent the period of the desired frequency. Therefore, a large number stored in the register corresponds to a low frequency tone. Also note that these division counters create a *linear* span of frequencies which have nothing to do with musical notes. The musical scale is based upon a *logarithmic* group of frequencies. Notes are arranged in *octaves* which represent a doubling of frequency and each octave is then subdivided into twelve equally tempered notes. Thus, every note is *the twelfth root of two* higher in frequency than the previous one. Table 7–1 shows the actual frequencies associated with each musical note along with the values that must be stored in the tuning registers to approximate them. Note that these are only approximations as can be seen in the last column which shows the actual frequencies produced by the T/S 2068.

Table 7–1. Register Values for Notes of Musical Scale

		Ideal	Tune Registers		Actual
Note	Octave	Frequency	Coarse	Fine	Frequency
C	1	32.703	13	43	32.705
C#	1	34.648	12	110	34.648
D	1	36.708	11	187	36.713
D#	1	38.891	11	19	38.889
E	1	41.203	10	116	41.200
F	1	43.654	9	222	43.646
F#	1	46.249	9	80	46.246
G	1	48.999	8	202	49.000
G#	1	51.913	8	76	51.907
A	1	55.000	7	213	54.988
A#	1	58.270	7	100	58.272
B	1	61.735	6	250	61.730
C	2	65.406	6	150	65.391
C#	2	69.296	6	55	69.296
D	2	73.416	5	222	73.402
D#	2	77.782	5	137	77.805
E	2	82.407	5	58	82.399
F	2	87.307	4	239	87.292

Table 7–1 — cont. Register Value for Notes of Musical Scale

		Ideal	Tune Registers		Actual
Note	Octave	Frequency	Coarse	Fine	Frequency
F#	2	92.499	4	168	92.492
G	2	97.999	4	101	98.000
G#	2	103.826	4	38	103.814
A	2	110.000	3	234	110.030
A#	2	116.541	3	178	116.543
B	2	123.471	3	125	123.460
C	3	130.813	3	75	130.783
C#	3	138.591	3	28	138.505
D	3	146.832	2	239	146.804
D#	3	155.563	2	197	155.501
E	3	164.814	2	157	164.798
F	3	174.614	2	119	174.723
F#	3	184.997	2	84	184.983
G	3	195.998	2	51	195.826
G#	3	207.652	2	19	207.627
A	3	220.000	1	245	220.060
A#	3	233.082	1	217	233.087
B	3	246.942	1	190	247.197
C	4	261.626	1	165	261.876
C#	4	277.183	1	142	277.010
D	4	293.665	1	119	294.000
D#	4	311.127	1	98	311.441
E	4	329.628	1	78	330.090
F	4	349.228	1	60	348.892
F#	4	369.994	1	42	369.966
G	4	391.995	1	25	392.349
G#	4	415.305	1	9	416.038
A	4	440.000	0	251	439.243
A#	4	466.164	0	237	465.190
B	4	493.883	0	223	494.395
C	5	523.251	0	211	522.512
C#	5	554.365	0	199	554.020
D	5	587.330	0	188	586.436
D#	5	622.254	0	177	622.881

Table 7–1 — cont. Register Value for Notes of Musical Scale

		Ideal	Tune Registers		Actual
Note	Octave	Frequency	Coarse	Fine	Frequency
E	5	659.255	0	167	660.180
F	5	698.456	0	158	697.785
F#	5	739.989	0	149	739.933
G	5	783.991	0	141	781.915
G#	5	830.609	0	133	828.947
A	5	880.000	0	125	882.000
A#	5	932.328	0	118	934.322
B	5	987.767	0	112	984.375
C	6	1046.502	0	105	1050.000
C#	6	1108.731	0	99	1113.636
D	6	1174.659	0	94	1172.872
D#	6	1244.508	0	89	1238.764
E	6	1318.510	0	84	1312.500
F	6	1396.913	0	79	1395.570
F#	6	1479.978	0	74	1489.865
G	6	1567.982	0	70	1575.000
G#	6	1661.219	0	66	1670.455
A	6	1760.000	0	63	1750.000
A#	6	1864.655	0	59	1868.644
B	6	1975.533	0	56	1968.750
C	7	2093.005	0	53	2080.189
C#	7	2217.461	0	50	2205.000
D	7	2349.318	0	47	2345.745
D#	7	2489.016	0	44	2505.682
E	7	2637.021	0	42	2625.000
F	7	2793.826	0	39	2826.923
F#	7	2959.956	0	37	2979.730
G	7	3135.964	0	35	3150.000
G#	7	3322.438	0	33	3340.909
A	7	3520.000	0	31	3556.452
A#	7	3729.310	0	30	3675.000
B	7	3951.067	0	28	3937.500
C	8	4186.009	0	26	4240.385
C#	8	4434.922	0	25	4410.000

Table 7–1 — cont. Register Value for Notes of Musical Scale

Note	Octave	Ideal Frequency	Tune Registers Coarse	Tune Registers Fine	Actual Frequency
D	8	4698.637	0	23	4793.478
D#	8	4978.032	0	22	5011.364
E	8	5274.041	0	21	5250.000
F	8	5587.652	0	20	5512.500
F#	8	5919.911	0	19	5802.632
G	8	6271.927	0	18	6125.000
G#	8	6644.876	0	17	6485.294
A	8	7040.000	0	16	6890.625
A#	8	7458.621	0	15	7350.000
B	8	7902.133	0	14	7875.000

By the way, you might like to compare Table 7–1 with the chart on pages 187-189 of the *T/S 2068 User Manual.* You will find that the frequencies given and the register values recommended are slightly different. The reasons for this are threefold. First, it seems that the actual frequencies presented in the Timex manual were calculated on a machine with limited accuracy. Therefore, there are some discrepancies in the last couple of decimal places. Next, when calculating the Coarse and Fine Tune register values, a couple more errors crept in. As evidenced by the program later in the chapter, we see that an approximation of 1.75 MHz was used for the clock frequency. We also see that the Fine Tune register value was truncated instead of being rounded to the nearest integer. Since all of the values are really approximations to the correct frequencies anyway, we really should use all of the accuracy we can get. Listing 7–1 shows the program that generated the values in Table 7–1.

Noise Generator

Aside from the three programmable tone generators, there is also a programmable noise generator in the PSG. This generates a random collection of frequencies that can be used in a variety of ways. Unlike a true white noise source which would generate all audio frequencies with equal amplitudes, this generator only produces a limited noise range. This range can be moved towards either the high-frequency or low-frequency end. This is accomplished by setting a five-bit value into

Listing 7–1

```
 1Ø   LET x=1764ØØØ/16
 2Ø   LET b=27.5: LET s=2↑ (1/12)
 3Ø   FOR i=3 TO 98
 4Ø   LET a=(b*s↑i)
 5Ø   GO SUB 5ØØ
 6Ø   PRINT a$;" ";
 7Ø   LET a=x/a: LET c=INT (a/256)
 8Ø   LET f=INT (a-(c*256)+.5)
 9Ø   IF c<1Ø THEN PRINT " ";
 91   PRINT c;" ";
1ØØ   IF f<1Ø THEN PRINT " ";
1Ø1   IF <1ØØ THEN PRINT" ";
1Ø3   PRINT f;" ";
11Ø   LET a=x/(c*256+f)
12Ø   GO SUB 5ØØ
13Ø   PRINT a$
4ØØ   NEXT i
499   STOP
5ØØ   LET a$=" "+STR$ (a+.ØØØ5):
FOR   j=1 TO LEN (a$): IF a$(j)=" ." THEN GO TO 5Ø3
5Ø1   NEXT j
5Ø2   LET a$=a$+"."
5Ø3   LET a$=(a$+"ØØØ") (j-4 TO j+3)
5Ø4   RETURN
```

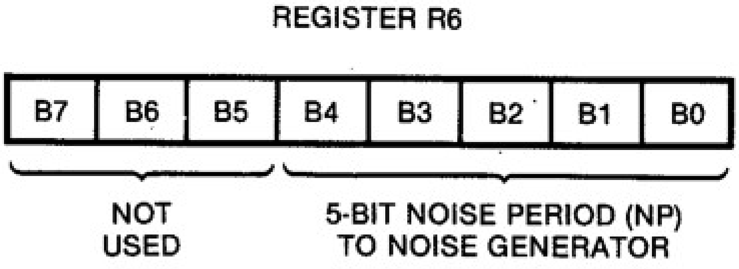
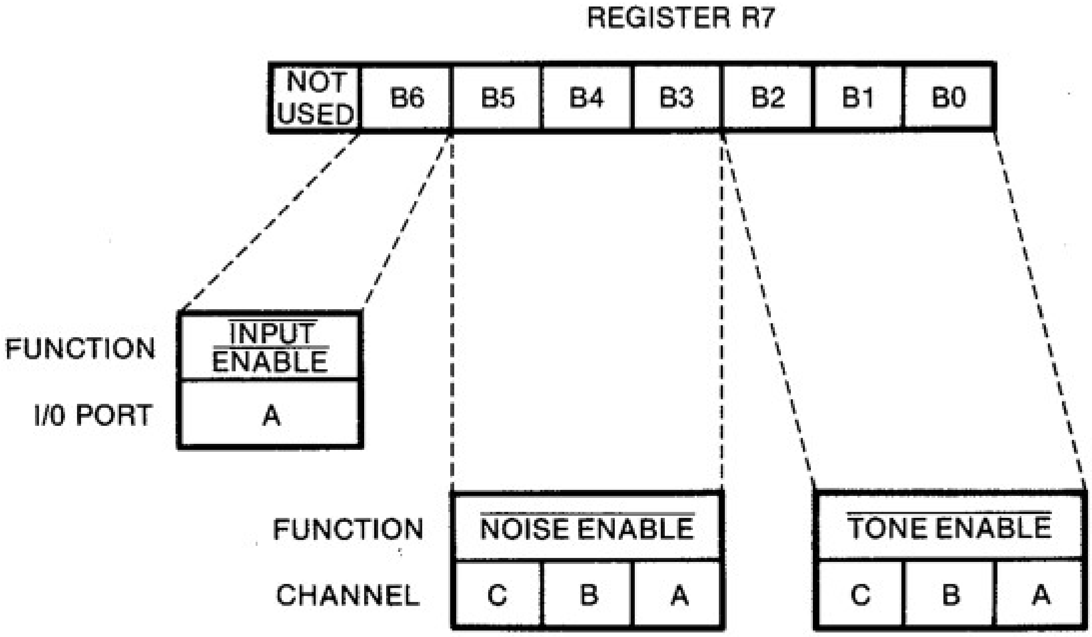


Fig. 7–5. Noise Period Register. (*Courtesy General Instrument Corp.*)

the Noise Period Register (R6). See Fig. 7–5. Using a low number gives a high-frequency noise that sounds somewhat like rain. A high number yields the lower pitched noise of wind rustling through the trees. By sweeping the noise spectrum from high to low, you can create the sound of waves crashing on the shore.

Selecting the Tone and Noise Sources

The Mixer Control Register (R7) takes care of selecting which tone or noise source will be sent to the output channels. Each channel (A, B, or C) can contain either its respective tone generator or the noise source. The tone enable is represented by the lower three bits of R7. The noise enable is contained in the next three higher bits. Each "0" bit represents that the respective source is enabled or turned on. Storing a value of 63 at this location (111111 in binary) causes all sound from the PSG to be turned off. Bit 6 is used to control the direction (i.e., either input or output) of an I/O port which is part of the PSG. This is the port used by the T/S 2068 to read the status of the external joysticks. This bit must always be left at zero for the STICK command to work properly. Fig. 7–6 shows the breakdown of bits in R7.



NOISE ENABLE TRUTH TABLE:

R7 BITS			NOISE ENABLED ON CHANNEL		
B5	B4	B3			
0	0	0	C	B	A
0	0	1	C	B	—
0	1	0	C	—	A
0	1	1	C	—	—
1	0	0	—	B	A
1	0	1	—	B	—
1	1	0	—	—	A
1	1	1	—	—	—

TONE ENABLE TRUTH TABLE:

R7 BITS			TONE ENABLED ON CHANNEL		
B2	B1	B0			
0	0	0	C	B	A
0	0	1	C	B	—
0	1	0	C	—	A
0	1	1	C	—	—
1	0	0	—	B	A
1	0	1	—	B	—
1	1	0	—	—	A
1	1	1	—	—	—

Fig. 7–6. Mixer Control Register. (*Courtesy General Instrument Corp.*)

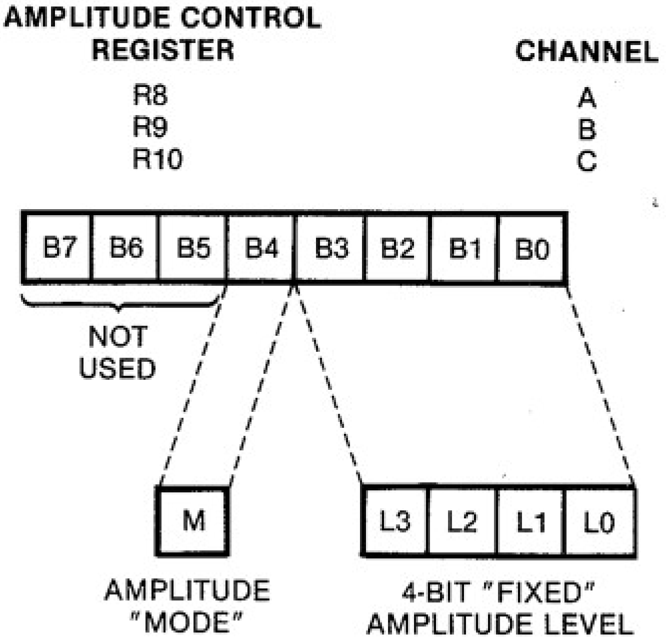


Fig. 7–7. Amplitude Control Registers. (*Courtesy General Instrument Corp.*)

Setting the Amplitude

Now that we have the frequency of the note(s) defined in the appropriate tune registers, we need to set the amplitude. There is an Amplitude Control register in the PSG for each of the three channels. These are R8, R9, and R10 for Channels A, B, and C, respectively. Each Amplitude Control register contains a four-bit "fixed" amplitude level and a one-bit amplitude "mode" control, as shown in Fig. 7–7.

The amplitude mode bit selects either a fixed level (M=0) or variable level (M=1) amplitude. If the fixed mode is selected, then the four-bit amplitude value sets the loudness of the channel to one of 16 preset levels. These levels are logarithmically related to compensate for the response of the human ear. The amplitude of a channel could be varied under software control by constantly writing new values into the amplitude register. The PSG, however, has a built-in mechanism for creating variable amplitude signals.

When the amplitude mode bit is set to a "1", another section of the PSG, called the *Envelope Generator* is used to control the channel's amplitude. The Envelope Generator supplies a four-bit value to the amplitude control which can dynamically change without any intervention of the CPU. At any given time, the output of the Envelope Generator will depend upon the Envelope Period Control Registers (R11 and R12) and the Envelope Shape/Cycle Control Register (R13). See Fig. 7–8 and Fig. 7–9.

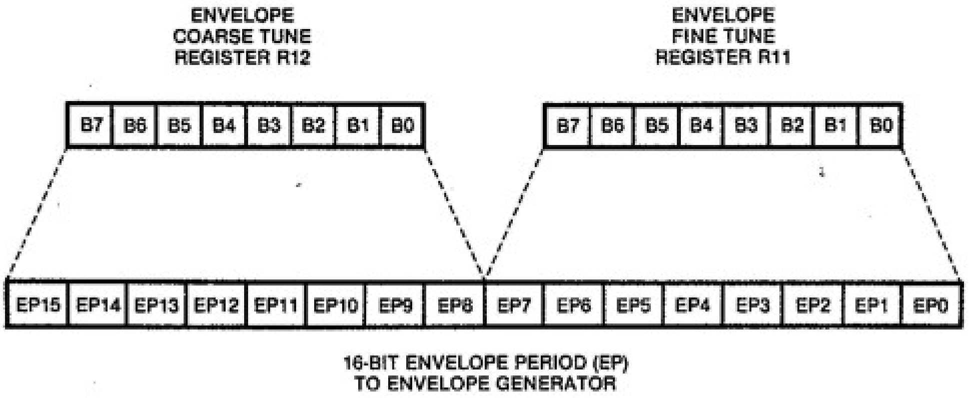


Fig. 7–8. Envelope Period Control Registers. (*Courtesy General Instrument Corp.*)

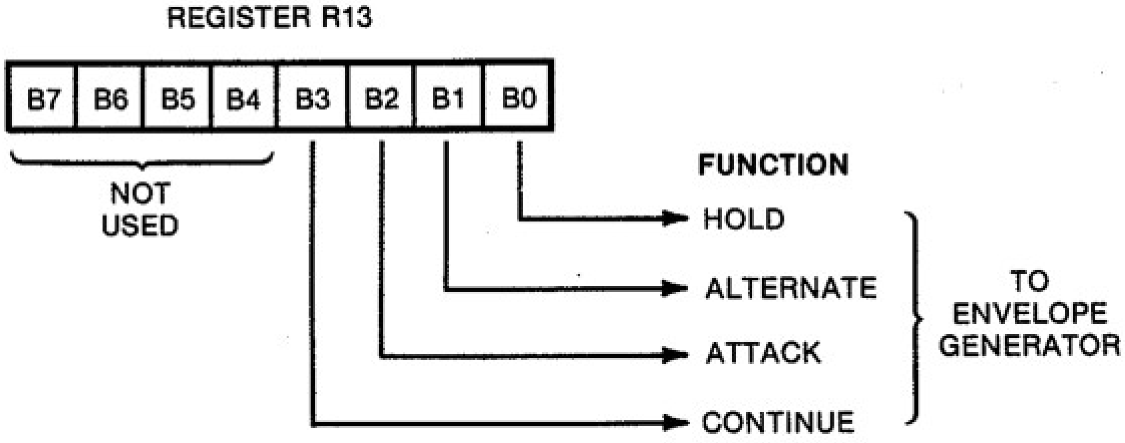


Fig. 7–9. Envelope Shape/Cycle Control Register. *(Courtesy General Instrument Corp.)*

The rate at which the amplitude will rise (attack) or fall (decay) is controlled by the Envelope Period Control Registers. These two registers combine to form a 16-bit divisor which determines the envelope period. Before being fed into this programmable divider, however, the master clock is first divided again by 16. Thus the envelope frequency can vary from 1.764 ÷ 16 ÷ 16 ÷ 1 = 6890 Hz down to 1.764 ÷ 16 ÷ 16 ÷ 65535 = 0.105 Hz. This last value corresponds to a decay period of 9.5 seconds.

Register 15 contains four bits which control the action of the Envelope Generator. These bits are labelled Hold, Alternate, Attack, and Continue. By setting these bits at the appropriate time, we can generate any type of attack-sustain-decay amplitude envelope that we desire.

The Hold bit determines whether the envelope control will generate a single attack or decay cycle or whether it will keep repeating. With

Hold = 1, the envelope will rise or fall — depending upon the value of the Attack bit — and then maintain the last amplitude (either all the way on or all the way off) until the next command. When Hold = 0, the envelope will continue cycling up and down at the frequency determined by the Envelope Period registers. The Alternate bit controls how the envelope cycles when Hold = 0. If the Alternate = 1, then the envelope will change direction at the end of each ramp cycle. That is, if the envelope started as an attack, it will begin to decay after reaching its maximum amplitude. This generates an envelope pattern that resembles a *triangle wave.* At the end of an attack cycle where Alternate = 0, the amplitude will return to zero and another attack cycle begins. This is known as a *sawtooth* pattern.

The Attack bit determines whether the envelope generated will be an attack (Attack = 1) or a decay (Attack = 0). The Continue bit actually does not serve any useful purpose so it is best left as a zero. Fig. 7–10 shows the resulting envelope shapes generated by the various settings of the Envelope Control bits.

COMBINING THE OUTPUT CHANNELS

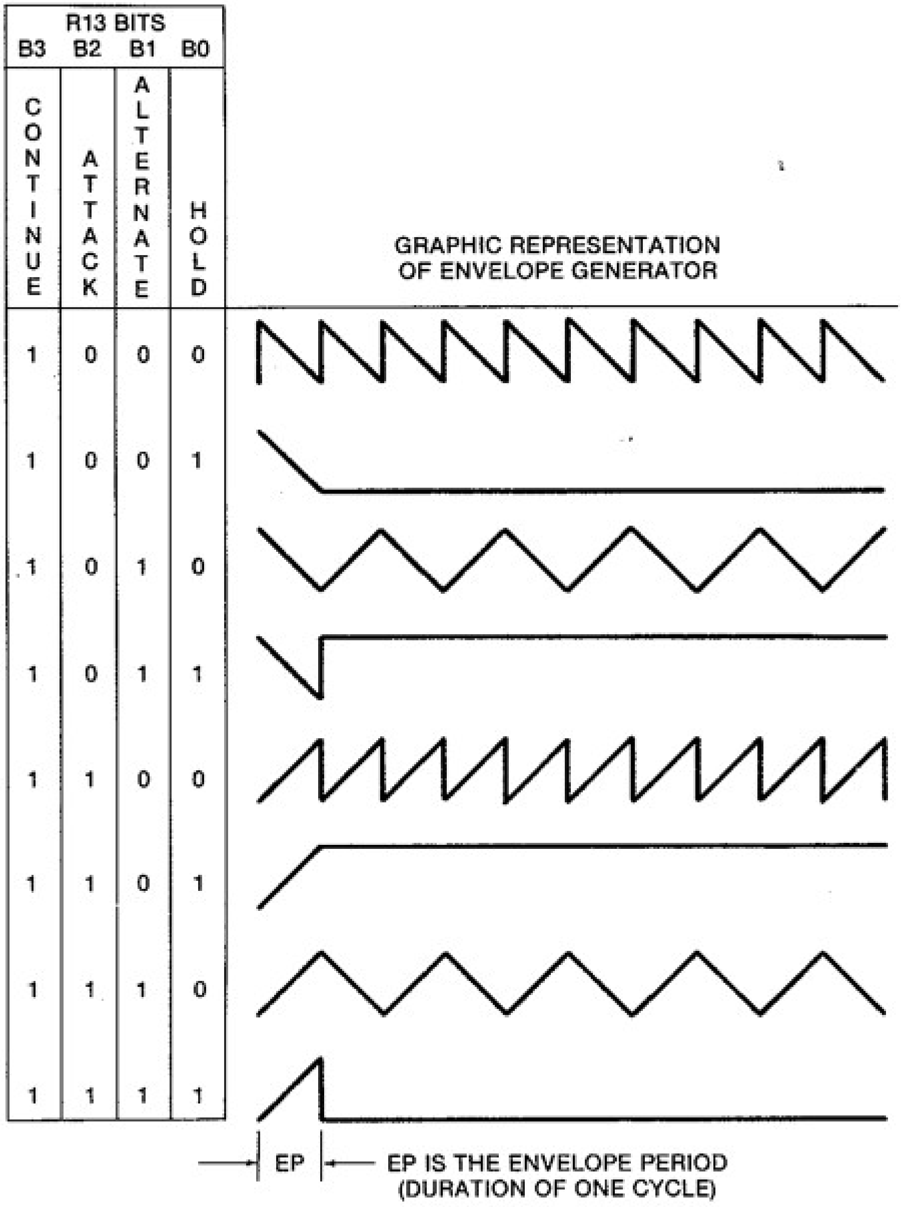
The three outputs from the PSG are combined into a single SOUND signal. This signal is available at the rear expansion connector and is one of three signals that drive the speaker in the T/S 2068.

OTHER EFFECTS

Tremolo

There are several musical effects obtainable from the PSG. One of these is called *tremolo,* which is the slight, rhythmical variation in a tone's amplitude. This is accomplished by cycling through several fixed amplitudes at a constant rate. A program loop can create such a cycle, with the execution time of the loop controlling the repetition, or tremolo *rate.* The range of amplitudes used will control the *depth* of tremolo effect. By using just 2 or 3 values, there will be only a slight warble effect. Increasing the range causes the effect to be more pronounced. Listing 7–2 contains a program to demonstrate the tremolo effect.

ENVELOPE SHAPE/CYCLE CONTROL



R13 BITS			
B3	B2	B1	B0
CONTINUE	ATTACK	ALTERNATE	HOLD
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

GRAPHIC REPRESENTATION OF ENVELOPE GENERATOR
EP
EP IS THE ENVELOPE PERIOD (DURATION OF ONE CYCLE)

Note: Above waveforms only show positive side of envelope.
For example, cycle 1110 creates a signal like this:

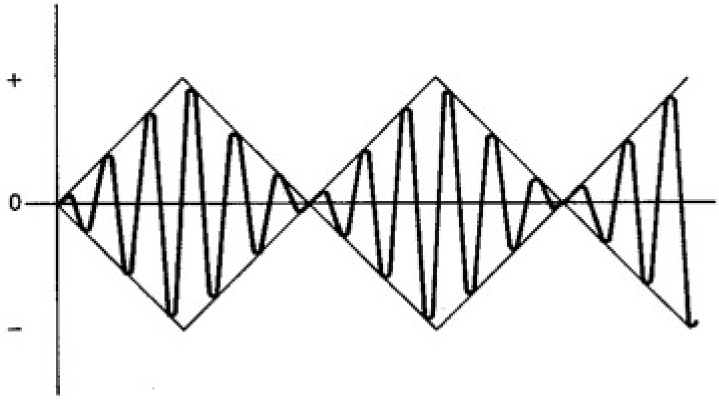


Fig. 7–10. Envelope patterns available with the PSG. (*Courtesy General Instrument Corp.*)

Listing 7–2

```
  1Ø   LET a=13
  2Ø   SOUND Ø,211;1,Ø: REM C5note
  3Ø   SOUND 7,62;8,13:        REM enable
  4Ø   PAUSE 1ØØ: REM play note
  5Ø   FOR i=1 TO 8: READ t: SOUND 8,a+t: REM do tremolo
  6Ø   NEXT i: RESTORE : GO TO 5Ø
 1ØØ   DATA Ø,1,2,1,Ø,-1,-2,-1
```

Vibrato

Varying the frequency of a note instead of its amplitude creates the effect known as *vibrato.* The program in Listing 7–3 demonstrates this effect. Note that the entire tone period value must be varied. Therefore, if the fine tune register must cross through a value of 0 or 255, the coarse tune register must be updated accordingly. To be equally effective on any given note, the depth of vibrato (lowest tone period - highest tone period) should be proportioned to the absolute period. The program in Listing 7–4 compares the tremolo and vibrato effects.

Listing 7–3

```
  1Ø   LET f=211:        REM C5 note
  2Ø   SOUND Ø,f;1,Ø
  3Ø   SOUND 7,62;8,13: REM enable
  4Ø   PAUSE 1ØØ:        REM play note
  5Ø   FOR i=1 TO 8: READ v: SOUND Ø,f+v: REM do vibrato
  6Ø   NEXT i: RESTORE : GO TO 5Ø
 1ØØ   DATA Ø,1,2,1,Ø,-1,-2,-1
```

Listing 7–4

```
  1Ø   LET a=13: LET f=211
  2Ø   SOUND Ø,f;1,Ø: REM C5note
  3Ø   SOUND 7,62;8,13: REM enable
  4Ø   PAUSE 1ØØ: REM play note
  45   FOR q=1 TO 5Ø
  5Ø   FOR i=1 TO 8: READ t: SOUND 8,a+t : REM do tremolo
  6Ø   NEXT i: RESTORE : NEXT q
  7Ø   FOR q=1 TO 5Ø
  8Ø   FOR i=1 TO 8: READ v: SOUND Ø,f+v : REM do vibrato
  9Ø   NEXT i: RESTORE : NEXT q: G O TO 1Ø
 1ØØ   DATA Ø,1,2,1,Ø,-1,-2,-1
```

Chords

Playing several notes at the same time can create the pleasant sound of a *chord.* Table 7–2 shows how various chords can be created. A chord starts with its *root* note (the numbers in parentheses refer to the octave: −2 means down one octave, −4 down two octaves, etc.). Then a 3rd Minor or 3rd Major note is added to create either a minor or major chord. Any of the remaining notes can then be added to fill out the chord. See Listing 7–5 for an example of how a chord is played on the T/S 2068.

Listing 7–5

```
 1Ø   SOUND Ø,211;1,Ø: REM C5
 2Ø   SOUND 2,167;3,Ø: REM E5
 3Ø   SOUND 4,141;5,Ø: REM G5
 4Ø   SOUND 7,56;8,15;9,15;1Ø,15
1ØØ   PAUSE 1ØØ
```

Table 7–2. Chord Selection Chart (Courtesy General Instrument Corp.)

Chord Selection	Root	3rd Minor	3rd Major	4th	5th	6th	7th
C	C (÷2)	D# (÷2)	E (÷2)	F (÷2)	G (÷2)	A (÷2)	A# (÷2)
C#	C# (÷2)	E (÷2)	F (÷2)	F# (÷2)	G# (÷2)	A# (÷2)	B (÷2)
D	D (÷2)	F (÷2)	F# (÷2)	G (÷2)	A (÷2)	B (÷2)	C (÷1)
D#	D# (÷2)	F# (÷2)	G (÷2)	G# (÷2)	A# (÷2)	C (÷1)	C# (÷1)
E	E (÷2)	G (÷2)	G# (÷2)	A (÷2)	B (÷2)	C# (÷1)	D (÷1)
F	F (÷2)	G# (÷2)	A (÷2)	A# (÷2)	C (÷1)	D (÷1)	D# (÷1)
F#	F# (÷4)	A (÷4)	A# (÷4)	B (÷4)	C# (÷2)	D# (÷2)	E (÷2)
G	G (÷4)	A# (÷4)	B (÷4)	C (÷2)	D (÷2)	E (÷2)	F (÷2)
G#	G# (÷4)	B (÷4)	C (÷2)	C# (÷2)	D# (÷2)	F (÷2)	F# (÷2)
A	A (÷4)	C (÷2)	C# (÷2)	D (÷2)	E (÷2)	F# (÷2)	G (÷2)
A#	A# (÷4)	C# (÷2)	D (÷2)	D# (÷2)	F (÷2)	G (÷2)	G# (÷2)
B	B (÷4)	D (÷2)	D# (÷2)	E (÷2)	F# (÷2)	G# (÷2)	A (÷2)

8
The Video Display

INTRODUCTION

When we interact with a computer, the primary output device through which information is conveyed is the video display. This is how the computer presents the results of most of its functions. The T/S 2068 has four different bit-mapped, color display modes available to the user. These will be described in this chapter.

VIDEO BASICS

It would seem that the best place to start is at the beginning — with a primer on what video is and how it works. In its broadest definition, *video* refers to the presentation of information as a visual image, usually by some electronic means. Nearly all video displays are created on some form of cathode-ray tube (crt), better known as a picture tube.

A crt consists of an electron gun assembly (see Fig. 8–1) and a relatively flat viewing area coated with a phosphorescent material. The electron gun contains a small wire filament that is heated by passing a current through it, much like an incandescent light bulb. This heating element gives off electrons which are attracted toward the face of the tube by a high voltage (generated by circuits within the television set or monitor). This voltage accelerates the electrons to a very high speed.

When the electrons strike the phosphors on the crt screen, they cause it to glow, creating a small speck of light. The intensity of this spot is controlled by an electrode which varies the number of electrons leaving the heater area. Finally, the location on the screen where the electrons hit can also be manipulated by deflecting the beam of electrons either electrostatically or magnetically. In most video displays, the deflection is done by means of two magnetic coils wound around the neck of the cathode-ray tube. These are known as the deflection coils. When the proper signals are applied to these coils, the electron beam can be directed left or right, up or down, at will.

With few exceptions, video displays use a *raster-scan* technique. This means that the electron beam is constantly being swept, in a consistent pattern, across the entire screen. Starting in the upper-left corner of the screen, the beam is made to scan towards the right. Wherever information is to be displayed, the electron (or beam) current is modulated so that it illuminates only the desired areas. When the beam reaches the far right side of the screen, it is turned completely off (or "blanked") and then rapidly returned to the left side in preparation for scanning the next line. At the same time the beam is also moving toward the bottom of the screen so that the next line will appear slightly below the previous one. When the beam eventually reaches the bottom of the screen, there's a similar blanking period during which the beam is brought back to its starting position.

To standardize the American television industry and to reduce the information needed to transfer a video image, the National Television System Committee (NTSC) was formed in 1953. This committee set

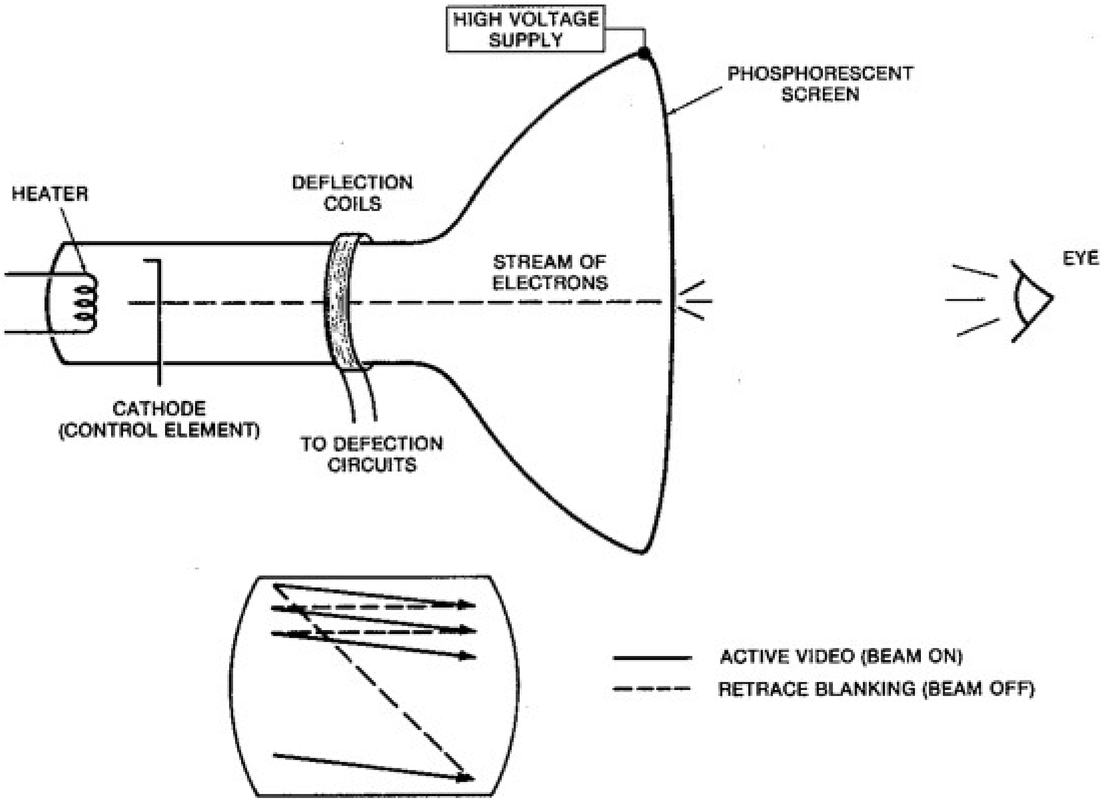


Fig. 8–1. Cathode-ray tube.

down the specifications for a raster-scan video system and further defined the frequencies, levels, and timing of the video signal.

It turns out that the electrical signal needed to deflect the electron beam horizontally and vertically in the appropriate fashion is quite easy to obtain. The waveform for this signal is known as a *sawtooth*, (so named for its resemblance to the ragged edge of a saw). Since this signal can be created by an oscillator circuit within the monitor or television, it need not be sent along with the video information from the camera, computer, or other video source.

In a camera, for instance, there might be another sawtooth oscillator that causes an electron beam to scan the face of a video pickup tube. The output of this tube would vary according to the brightness level of the scene at the exact location of the beam. Thus, if the output of the pickup tube were connected to the input of the crt, it would "paint" an image on the screen of whatever the camera was focused on.

There is, of course, one hitch. For the system to work properly, the scanning position of the pickup tube and the beam deflection of the crt must coincide with, and track, each other. Although the NTSC specification sets the frequency for these oscillators, some means of getting them "in phase" or *synchronized* must be provided. This is accomplished by adding some special features, known as *horizontal* and *vertical sync*, to the video signal. See Fig. 8–2 and Fig. 8–3.

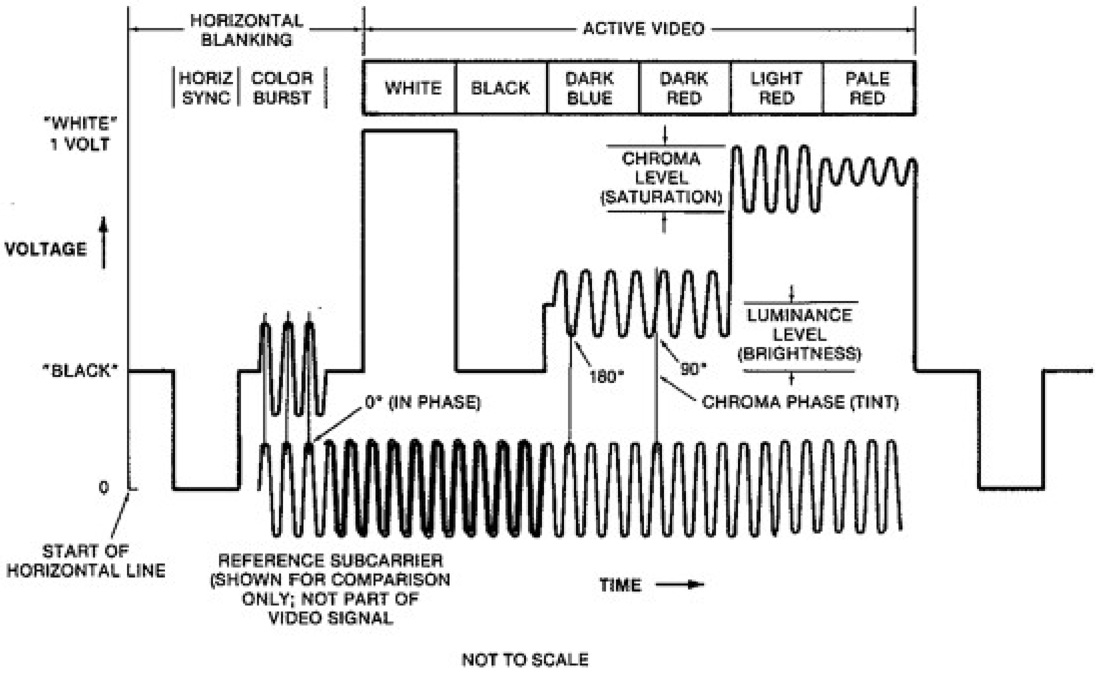


Fig. 8–2. Video waveform for one horizontal scan line.

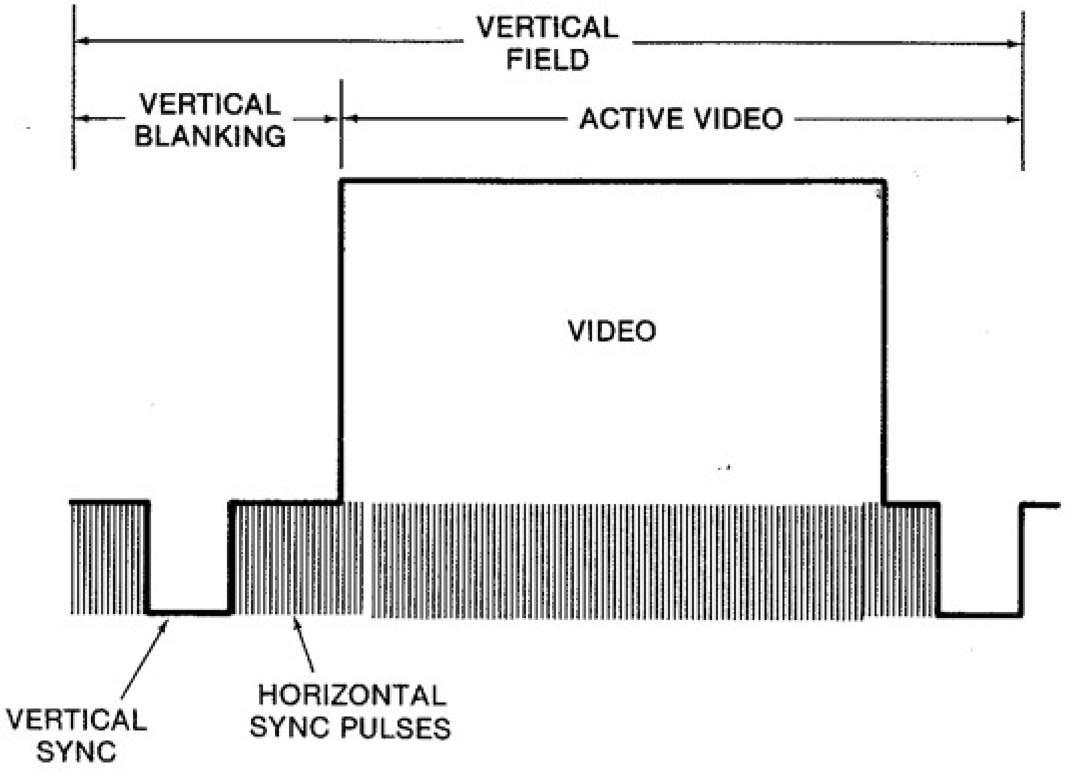


Fig. 8–3. Video waveform for one vertical field.

The preceding holds true for both black-and-white and color transmission. When color television was first considered by the NTSC, it was decided that some means of adding the color information that would remain compatible with black-and-white standards needed to be found. A color tv camera works by separating the light from a scene into its red, green, and blue components. It's like putting three black-and-white cameras together with a colored filter in front of each one. Since red, green, and blue are the primary colors, any other color can be broken down into a specific mixture of these colors.

To be compatible with the black-and-white video standard, the RGB (Red, Green, and Blue) signals must be combined in a precise ratio to develop the brightness, or *luminance* signal. By adding these three signals, we can create another signal which is identical to the output of a black-and-white camera. That takes care of the compatibility issue, but what about the rest of the color information? Well, it can be proven mathematically that a set of mutually exclusive functions (such as the RGB primary colors) can be transformed into another set based upon any other mutually exclusive functions. By adding the RGB signals together we have already created one signal. If we add or subtract the R, G, and B signals in two other ratios, we can derive another pair of signals that represent *only* the color information. With proper processing, these signals can then be transformed to represent the *hue* (tint) and *saturation* (color intensity) information.

The NTSC's solution was to add a high-frequency subcarrier to the luminance signal. This could be both amplitude and frequency modulated (actually phase modulated) to carry the hue and saturation information. This subcarrier would ride along the video waveform so that to a black-and-white receiver it would pass unnoticed. With the proper circuitry in a color set, however, this extra signal could be reconstituted into the proper red, green, and blue signals to drive a typical color crt.

As was the case with the horizontal and vertical sync signals, another *color sync* signal needed to be added so that the receiver's color oscillator could be synchronized to that of the original video source. This additional signal is called the *color burst* because it consists of a burst of approximately nine cycles of the originating subcarrier reference oscillator.

To summarize the NTSC color video system, let's consider its three separate parts. Both the sending and receiving devices contain three oscillators: a vertical one at a rate of 29.97 Hz, a horizontal one at 15,735 Hz, and a subcarrier riding on 3.57945 MHz. Through these oscillators, the sending device creates a video signal with the following properties:

1. There is a negative-going pulse to signify the start of each horizontal line and a similar signal to signify the beginning of each vertical field.
2. The color reference subcarrier is modulated by signals representing the red, green, and blue content of each portion of the active video field. These signals in turn create an am (amplitude modulated) and fm (frequency modulated) signal that rides along with the brightness or luminance signal. In particular, the phase of this subcarrier represents the hue, and the amplitude of the subcarrier corresponds to the saturation of the color. This subcarrier signal is referred to as the *chrominance* subcarrier, or simply *chroma.* The center point around which the subcarrier oscillates is thus the luminance level.

Now that we know what kind of signal the T/S 2068 video generator must create, we will look at where the computer stores the information that we see on the screen.

DISPLAY MODES

The information that the T/S 2068 displays on the screen comes from several special areas in RAM. These are called *display files* and *attribute files* and there are two of each. Their addresses are shown in Fig. 8–4. Normally, only the primary files are used, and the memory assigned to the secondary files is used by the system software. To fully utilize the secondary files, this system data must be moved elsewhere in RAM as discussed in Chapter 5.

Display and Attribute File Addresses	Hexadecimal	Decimal
Display File 1	4000-57FF	16384-22527
Attribute File 1	5800-5AFF	22528-23295
Display File 2	6000-77FF	24576-30719
Attribute File 2	7800-7AFF	30720-31487

Fig. 8–4. Display file and attribute file addresses.

The display file holds a *bit-mapped* representation of the video display. This means that there is a one-for-one correspondence between each bit in the display file memory and the dots that make up the video picture. Each dot is also called a picture element, or *pixel* for short. Although the exact order of the bits in memory may not correspond to the relative position on the screen, there is a somewhat logical layout to the bytes in the display file. See Fig. 8–5.

The attribute file contains information on the color, intensity, and flashing status for each character position in the display. The information represented by each attribute byte is shown in Fig. 8–6 and the layout of the attribute file is shown in Fig. 8–7. Note that three bits are used to represent each of the PAPER and INK colors. If you examine the bit representations, you will find that the least significant bit represents blue; the middle bit red; and the most significant bit represents green. Various combinations, of course, yield their respective colors (e.g., 011 = Red + Blue = Magenta). Thus the T/S 2068 represents color information in an RGB format.

A large portion of the T/S 2068's circuitry is devoted to the job of video generation. This is one of the major improvements that the T/S 2068 made over the earlier T/S 1000 which had used the CPU to perform this task. Since video generation requires that a continuous signal be created and updated about 60 times per second, it required

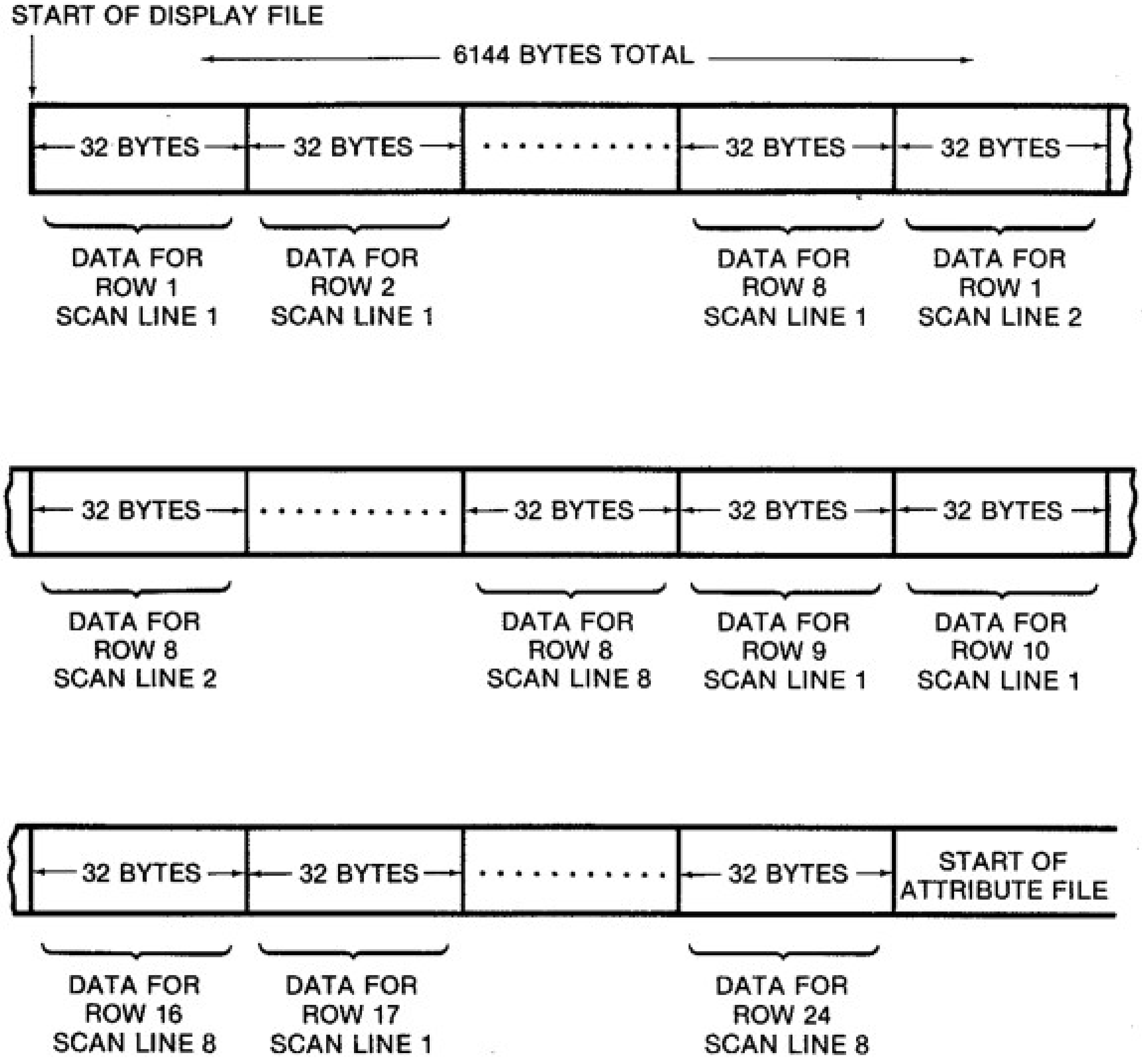
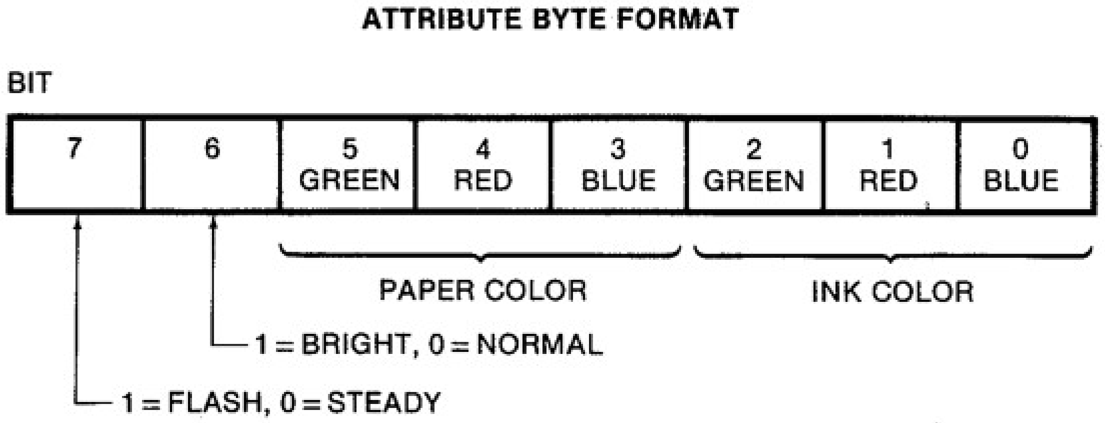

START OF DISPLAY FILE
6144 BYTES TOTAL
32 BYTES
32 BYTES
32 BYTES
32 BYTES
DATA FOR ROW 1 SCAN LINE 1
DATA FOR ROW 2 SCAN LINE 1
DATA FOR ROW 8 SCAN LINE 1
DATA FOR ROW 1 SCAN LINE 2
32 BYTES
32 BYTES
32 BYTES
32 BYTES
DATA FOR ROW 8 SCAN LINE 2
DATA FOR ROW 8 SCAN LINE 8
DATA FOR ROW 9 SCAN LINE 1
DATA FOR ROW 10 SCAN LINE 1
32 BYTES
32 BYTES
32 BYTES
START OF ATTRIBUTE FILE
DATA FOR ROW 16 SCAN LINE 8
DATA FOR ROW 17 SCAN LINE 1
DATA FOR ROW 24 SCAN LINE 8

Fig. 8–5. Display file organization. (*Courtesy Timex Computer Corp.*)

that the T/S 1000 either stop making video while it was computing, or else run very slowly.

In the T/S 2068 the video generator operates independently (but synchronously) with the CPU. To present information on the video display, the CPU has only to load the correct dot patterns into the display file and the appropriate color/intensity information into the attribute file. The video generator then constantly reads out these files, converting the information they contain into the three RGB signals. These signals are available at the card edge on the rear of the computer; this allows the T/S 2068 to be connected to monitors specifically designed to accept RGB signals. We'll have more to say about this later. The RGB signals also go to an *encoder* circuit which transforms them into the *composite* video signal which feeds the MONITOR jack and the RF modulator.



PAPER OR INK COLOR

VALUE		COLOR
7	111	WHITE
6	110	YELLOW
5	101	CYAN
4	100	GREEN
3	011	MAGENTA
2	010	RED
1	001	BLUE
0	000	BLACK

Fig. 8–6. Attribute byte structure. (*Courtesy Timex Computer Corp.*)

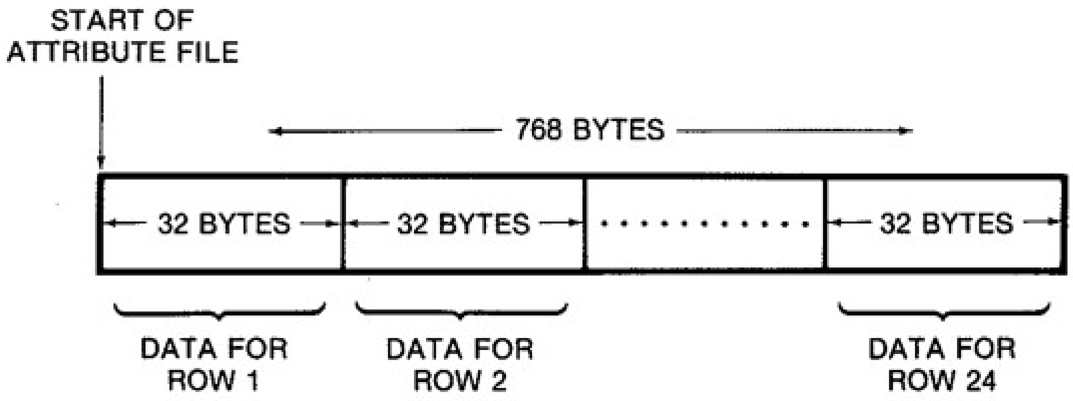


Fig. 8–7. Attribute file organization. (*Courtesy Timex Computer Corp.*)

Display Mode 1

In this mode, the T/S 2068 organizes the video display into 24 rows of 32 characters each. This corresponds to creating 256 pixels on each of 192 lines. One attribute byte is used for each character position (a character is made up of 8 × 8 pixels). The pixel data and attribute data are stored in their primary locations (D_FILE_1 and A_FILE_1).

Display Mode 2

This display mode allows the computer to generate 24 rows of 64 characters each. This corresponds to a graphic resolution of 512 × 192 dots. Both display files are used. The odd numbered character positions come from D_FILE_1 and the even ones come from D_FILE 2. No attribute file is used; therefore, the entire screen can only have one PAPER color and one INK color. The paper color is stored in a hardware register at port FFh. The ink color is chosen automatically by the computer as shown in Table 8–1. The low brightness and flashing attributes are not available in this mode.

Table 8–1. Paper and Ink Combinations Available in Display Mode 2

	Port FFh			
Bit5	Bit4	Bit3	Ink	Paper
0	0	0	Black	White
0	0	1	Blue	Yellow
0	1	0	Red	Cyan
0	1	1	Magenta	Green
1	0	0	Green	Magenta
1	0	1	Cyan	Red
1	1	0	Yellow	Blue
1	1	1	White	Black

You can see what the 64-column mode looks like by typing:

```
OUT 255,62
```

This turns on Mode 2 with black paper and white ink. You will need a good, hi-resolution monitor connected to the MONITOR jack in order to see the presentation clearly.

Note that the characters are now half as wide as normal. You can type on the computer, or do anything else with it as if you were in the regular mode. However, the upper portion of the screen will have strange looking "garbage" characters in every other position. In fact, as you press the keys, you will notice certain changing patterns on the screen. This is because the second display file memory is still being used by the system software for data storage. The only effect of the OUT 255,62 command has been to cause this area of memory to be displayed on the screen.

Display Mode 3

This is the "secondary page" mode of the T/S 2068. It operates exactly like display mode 1 except that D_FILE_2 and A_FILE_2 are used. One of the reasons for having two separate display pages is that they can be used for *instant reveal* and simple animation. Without two pages, the computer would have to constantly update a single page with the new information to be presented. Updating an entire screen takes a finite amount of CPU time, and you would have to watch the computer draw each new screen. With two pages, however, the computer can be updating one page while the other is being displayed. Then it can simply "throw a switch" to instantly reveal the newly drawn page and begin updating the first page. By alternating between these two pages, a crude form of animation can be created.

Display Mode 4

This is the Ultra High Color Resolution mode of the T/S 2068. It uses D_FILE_1 to define the pixel data just as in mode 1. Instead of using A_FILE_1 to hold the attributes, however, this mode uses D_FILE_2. Since this display file contains eight times as much memory as the normal attribute file, it is possible to assign an attribute byte to each row of pixels within each character. This allows a wide variety of color displays to be generated by the T/S 2068.

RGB VIDEO

As previously mentioned, the video generator first created Red, Green, and Blue signals from the data stored in the display/attribute file areas of RAM. These signals can be used to drive an RGB monitor for high quality color displays. By using the RGB signals directly, such a monitor avoids the noise and distortion that are inevitable when the RGB signals are converted to a composite video signal and then back again.

Most RGB monitors require five signals: the Red, Green, and Blue video signals plus Horizontal and Vertical sync. These last two signals are sometimes combined into a *composite sync* signal. These signals are usually sent in digital form as TTL levels (transistor-transistor-logic, the most common form of digital electronic circuitry using +5-volt and 0-volt signal levels). The T/S 2068 has TTL compatible R, G, and B signals available at its card edge. However, there is no horizontal, vertical, or composite sync signal available. To obtain these signals, it is necessary to feed the composite *video* signal (also available at the card edge) into a circuit that can extract the sync portion and translate it to TTL levels. If the monitor requires *positive* TTL sync, then the circuit must also invert the sync waveform since the composite video signal uses negative going sync pulses. An RGB adapter should be available by now from the Timex Computer Corporation. When using RGB, however, the intensity attribute (set by the BRIGHT instruction) is ignored.

PART 2
MACHINE LANGUAGE

SECTION A
PROGRAMMING THE Z80

9
Introduction to Machine Language Programming

We shall now examine how to program the Z80 in machine language. We have covered most of the hardware aspects of the CPU; now we'll explore the software side. The principles of machine language programming are not too different from BASIC programming. We must devise a procedure, or *algorithm*, to solve a problem and then turn that into a series of program steps within the constraints of the programming language. We'll examine these constraints a little further.

BASIC VS. MACHINE LANGUAGE

Every BASIC program written on the T/S 2068 follows a given set of rules. The program consists of lines or statements which themselves follow a given set of rules. These lines are executed sequentially unless specifically directed to jump or GO TO another line number. We define various data elements as either constants or variables in either numerical or character (i.e., string) format. Variables are referred to by "names" which are usually chosen to help remember their meaning. Within each statement, the data is manipulated using the approximately 100 different commands that are understood by the T/S 2068 BASIC interpreter. These commands include the simple arithmetic operations as well as higher mathematics such as trig functions. Thus writing a program is simply the act of stringing these instructions together in the proper order to produce the required results.

Machine language programs also follow a rigid set of rules. Again, we can define a program as a list of statements which execute sequentially unless explicitly diverted. As far as a machine language program is concerned, however, all data looks the same — it's simply a "bunch of bits." The Z80 doesn't know whether a given byte represents part of a floating point number or one character of a string. It is strictly up to the machine language programmer to keep track of things. One way to do this is by creating certain *data structures* that help define the program. We have already seen several examples of data structures used by the T/S 2068 BASIC interpreter: numerical values are stored in 5-byte blocks (fixed length data structure), string variables are stored in a variable length data structure, and the BASIC program in memory is stored as a linked-list. Keep in mind that these structures merely refer to blocks of memory and are identified to the Z80 by their absolute starting address. Actually, we can assign names to such variables or lists if we have an assembler (more on this later), but to the Z80 everything is still simply 8 bits times 65,536 locations. Finally, unlike the BASIC interpreter which understands about 100 commands, the Z80 has an "instruction set" of only 50 or so unique functions. (Most Z80 instructions have many variations, however.) These instructions are all very simple compared to their BASIC counterparts. For example, the Z80 only has instructions for adding and subtracting 8 and 16-bit numbers. All other mathematical operations require multiple Z80 instructions arranged in a small *routine* to accomplish the given function. Some examples of this will be shown in Chapter 12.

HAND-CODING VS. AN ASSEMBLER

Before delving into Z80 machine language programming, we must decide how we are going to write programs and get them into the T /S 2068. There are two ways to write programs: hand-coding or using an *assembler*. Both methods start off by defining the problem in terms of the Z80's capabilities. A sequence of Z80 instructions is then written down which will accomplish the desired task.

Each instruction is referred to by a *mnemonic* code. When hand-coding a program, the one or more bytes that represent this instruction, i.e., its *op code,* are looked up in a table and written down next to the instruction. Any related data bytes are also written down in the proper order to derive the exact sequence of bytes that represent the desired program.

Using an assembler makes things much simpler, however. The mnemonic program is typed directly into the assembler which automatically converts it into the proper bytes in memory. Various other features such as *line labels* and *variable labels* (similar to BASIC line numbers and variable names) are also supported. Another great feature of an assembler is in calculating relative offsets and jump locations which may change as the program is modified. As we will soon see, adding one instruction to an existing program can require many changes. With hand-coding, each change must be identified and corrected manually.

Despite the advantages of using an assembler, we are going to proceed with hand-coding our examples. The reason for this is quite simple. At the time this book was being written there was no assembler program available for the T/S 2068. Anyone serious about machine language programming, however, will eventually want to have an assembler (there should be some available by now). Although we don't have an assembler, we can still write programs in "assembly language" complete with instruction mnemonics and labels. Then we will go back and play human assembler — looking up op codes, calculating offsets, and eventually generating the hexadecimal bytes which represent the given program. To enter the program into the T/S 2068, we then have to convert the hex numbers into decimal and POKE them in from BASIC. While this takes much more time and effort than using an assembler, it offers a great opportunity to learn about the Z80 and how it is programmed.

UNDERSTANDING THE Z80 DOCUMENTATION

Our approach to learning machine language programming will center upon one theme — understanding completely the Z80 instruction set. The charts presented in the next few chapters outline everything you need to know about what each instruction does.

Just to get a feel for the usefulness of these charts, take a look at the column headings in Table 10–1. The first column shows the mnemonic for the given instruction. This includes separate representations for the various types of operands that can be used with each instruction. Due to the variety of addressing modes available with the Z80, there may be twenty or more different forms of a given instruction. The eight-bit LD (load) instruction, for example, has 21 varieties.

The next column shows a symbolic representation of the operation performed by the instruction. Arrows are used to denote the flow of data. The next eight columns show the effect of the instruction on each bit of the flag register. There are five distinct possibilities for the outcome of each bit. By executing the instructions, a flag bit can be "set" (=1) or "reset" (=0), left unaffected, or left in an unknown state. The last alternative is that the flag bit is set or reset according to the result of the operation performed by the instruction.

The next two columns give the numeric representation of the instruction, or *op code,* in both binary and hex form. Many instructions apply to more than one register so a generic form is listed for the op code. Substituting the proper bits for each of the required registers will yield the exact binary representation for the instruction.

The last three columns give information on the number of bytes used by each instruction and the number of machine cycles needed to execute it. Normally, these columns can be ignored unless a very time-sensitive routine is being written.

THE USR COMMAND

To execute a machine language program from BASIC, the USR command is used. This is a cross between the BASIC GO SUB statement and the machine language CALL instruction. Actually, USR is a *function* so it requires a preceding action command such as PRINT or LET. It also requires a parameter which in this case is the address to begin executing at (in decimal). Thus, a machine language routine located at address 1000 can be executed by any one of these lines:

```
10 LET x = USR 1000
20 PRINT USR 1000
30 RANDOMIZE USR 1000
```

All of the preceding statements cause a machine language routine to execute until a RETurn instruction is encountered. This returns control to the BASIC program which then continues with its next statement. Upon entering the machine language routine, the BC register pair will hold the starting address of that routine. With the exception of the IY and I registers, the routine is free to use any of the CPU registers. Their values prior to the USR call are automatically saved on the machine stack. Upon returning to BASIC, the interpreter will restore these registers to their proper value. The BC register pair, however, can be used to pass back data from the machine language routine. The contents of this register will be assigned as the value of the USR function. Thus, in line 10, the value in register BC will be assigned to the variable x. This is a full 16-bit integer value.

In line 20, the value returned will be printed out on the screen. Line 30 shows a method of calling a routine which does not pass back any data. This avoids the need for a "dummy variable" or the printing of extraneous numbers on the screen.

THE Z80 INSTRUCTION SET

The remainder of this section will describe each of the Z80 instructions in detail. Chapter 10 will introduce the simple LOAD instructions for moving information between memory and the CPU. All of the Z80 addressing modes will be described. Chapter 11 continues with instructions that affect program flow. These include jump, call, and restart instructions. Chapter 12 will discuss arithmetic and logic operations. Routines for multiple precision addition, subtraction, and multiplication are described, as well as a routine to perform division. Finally, Chapter 13 concludes with the advanced Z80 instructions, including block moves, searches, and I/O operations.

10
Moving Information: The Load Instruction

Machine language programs spend much of their time transferring data between the CPU registers and memory. Moving data is the function of the Z80's LOAD instructions which are very similar to the LET statement in BASIC. There are many varieties of this instruction, so it will be helpful to break them down into at least two groups: the 8-bit LOAD group and the 16-bit LOAD group. As its name implies, the 8-bit load group transfers one byte of data per instruction. Note that whenever we speak of the transfer of data, we are actually talking about reading the contents of some *source* location and then copying the same pattern of bits into the *destination* location. These locations can be CPU registers, RAM or ROM memory locations, or even I/O ports. It is also important to note that the contents of the source location always remain unchanged (unless, of course, the source location also happens to be the destination location). The previous contents of the destination location are always lost when the new data is written there.

REGISTER ADDRESSING

Table 10–1 shows the 8-bit LOAD group of the Z80. All of the instructions start with the LD mnemonic which is short for LoaD. Starting at the top, we have the LD r,s instruction. Here r and s refer to any of the following registers within the Z80: A,B,C,D,E,H, or L. As the symbolic description to the right implies, the contents of register *s* are copied into register *r*. Since both the source and destination locations are registers, this instruction is called a register to register load. Register addressing is relatively simple (there are only seven to choose from) and extremely fast. Since it only takes 3 bits each to

Table 10–1. The 8-Bit LOAD Group (Courtesy MOSTEK, Corp.)

Mnemonic	Symbolic Operation	Flags								Op-Code		No. of Bytes	No. of M Cycles	No. of T States	Comments	
		S	Z		H		P/V	N	C	76 543 210	Hex					
LD r, s	r ← s	•	•	X	•	X	•	•	•	01 r s		1	1	4	r, s	Reg.
LD r, n	r ← n	•	•	X	•	X	•	•	•	00 r 110		2	2	7	000	B
										← n →					001	C
LD r, (HL)	r ← (HL)	•	•	X	•	X	•	•	•	01 r 110		1	2	7	010	D
LD r, (IX+d)	r ← (IX+d)	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19	011	E
										01 r 110					100	H
										← d →					101	L
LD r, (IY+d)	r ← (IY+d)	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19	111	A
										01 r 110						
										← d →						
LD (HL), r	(HL) ← r	•	•	X	•	X	•	•	•	01 110 r		1	2	7		
LD (IX+d), r	(IX+d) ← r	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19		
										01 110 r						
										← d →						
LD (IY+d), r	(IY+d) ← r	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19		
										01 110 r						
										← d →						
LD (HL), n	(HL) ← n	•	•	X	•	X	•	•	•	00 110 110	36	2	3	10		
										← n →						
LD (IX+d), n	(IX+d) ← n	•	•	X	•	X	•	•	•	11 011 101	DD	4	5	19		
										00 110 110	36					
										← d →						
										← n →						
LD (IY+d), n	(IY+d) ← n	•	•	X	•	X	•	•	•	11 111 101	FD	4	5	19		
										00 110 110	36					
										← d →						
										← n →						
LD A, (BC)	A ← (BC)	•	•	X	•	X	•	•	•	00 001 010	0A	1	2	7		
LD A, (DE)	A ← (DE)	•	•	X	•	X	•	•	•	00 011 010	1A	1	2	7		
LD A, (nn)	A ← (nn)	•	•	X	•	X	•	•	•	00 111 010	3A	3	4	13		
										← n →						
										← n →						
LD (BC), A	(BC) ← A	•	•	X	•	X	•	•	•	00 000 010	02	1	2	7		
LD (DE), A	(DE) ← A	•	•	X	•	X	•	•	•	00 010 010	12	1	2	7		
LD (nn), A	(nn) ← A	•	•	X	•	X	•	•	•	00 110 010	32	3	4	13		
										← n →						
										← n →						
LD A, I	A ← I	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9		
										01 010 111	57					
LD A, R	A ← R	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9		
										01 011 111	5F					
LD I, A	I ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9		
										01 000 111	47					
LD R, A	R ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9		
										01 001 111	4F					

Notes: r, s means any of the registers A, B, C, D, E, H, L
IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

specify the source and destination registers, the entire instruction can be contained within a single byte and executed in one machine cycle.

To see an example of how this instruction would be used, suppose we have some data in the A register that we wish to move into the B register. The mnemonic code would be:

```
LD B,A
```

To calculate the binary representation of the op code, we start with 01, add 000 to specify B as the destination, and finish up with 111 indicating A for the source. This gives us an op code of 01000111 or 47h. You can also verify this by checking the op code chart in Table 10–2. Locate the column for register A as the source and the row for register B as the destination. The intersection gives 47 as the correct op code. You should also note that none of the flag bits are affected by this instruction.

Table 10–2. Op Codes for 8-Bit LOAD Group (Courtesy MOSTEK, Corp.)

		SOURCE															
		IMPLIED		REGISTER							REG INDIRECT			INDEXED		EXT. ADDR.	IMME.
		I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX + d)	(IY + d)	(nn)	n
DESTINATION REGISTER	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
REG INDIRECT	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			02													
	(DE)			12													
INDEXED	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
EXT. ADDR	(nn)			32 n n													
IMPLIED	I			ED 47													
	R			ED 4F													

IMMEDIATE ADDRESSING

The next form of LoaD instruction to consider is that shown by the following mnemonic:

```
LD r,n
```

Here, as before, *r* represents one of seven possible registers for the destination. In this case, however, the source of data, *n*, is the next byte following the op code. Therefore, this instruction requires two bytes — one for the op code and another for the data. When this instruction is executed, the Z80 looks at the second byte and then copies that data into the selected register.

This method of supplying the data within the instruction is called *immediate addressing.* That is, the source data is found immediately following the op code. For example, if we wanted to load register C with a value of 40, the instruction would look like:

```
    ØE  28  LD C,4Ø
```

Notice that we have written some (hex) numbers in front of the instruction. These are the hexadecimal bytes that represent the given instruction. This format of writing the hex notation to the left of the mnemonic representation is standard for an assembly language listing. Even though we will be assembling our code by hand, it proves to be a convenient and efficient way of showing both forms at one time. Therefore, whenever writing machine language code, leave enough space to fill in the hex codes to the left as they are determined.

To see how these hex numbers were derived, start by looking up the op code for loading register C with an immediate value. From Table 10–2 we find that the code should read 0E n (where n is the data). Thus the first byte must be 0Eh. The instruction specified a data value of 40 without explicitly indicating its base. Therefore, it is assumed to be a decimal number which needs to be converted into hex before it can become the second byte of the instruction. Doing the conversion gives us 28h for the second byte. Therefore, the complete hex representation for the instruction is 0E 28 as shown.

REGISTER INDIRECT ADDRESSING

The next entry in the 8-bit LOAD group demonstrates the use of *register indirect* addressing. Indirect addressing refers to a method whereby the data resides in a memory location, and the address of that location is placed in one of the 16-bit register pairs within the Z80. The CPU, therefore, must look to that address to find the data. (This may sound a bit confusing, but it is a very common procedure for the Z80 and can be very powerful.)

Of course, to use any of the register indirect load instructions, we must first set up one of the register pairs with the desired address. Note from Table 10–2 that any register can be loaded using the (HL) indirect addressing mode. The other register pairs, however, can only be used for loading the A register.

For example, if we needed to load register D with the contents of memory location 1000h we would write:

```
LD H,10h
LD L,00h
LD D,(HL)
```

See if you can fill in what the correct hex codes should be. When we discuss the 16-bit LOAD group, we will find an even easier way to load the HL register pair. Remember that whenever we use parentheses around a 16-bit value, we are actually referring to the 8-bit data stored at the 16-bit address. Therefore, you should read *HL* as the value stored in the HL register; but *(HL)* means "the value stored at the address pointed to by the HL register."

INDEXED ADDRESSING

One of the most powerful addressing modes of the Z80 is known as *indexed addressing*. The mode makes use of the two special purpose, 16-bit, index registers: IX and IY. While these registers are capable of holding a complete memory address, the term indexing refers to the fact that we can add an offset, or displacement, to the address to form a new *effective address*. This effective address is then used as the source or destination location for the data.

One of the most common uses for indexed addressing is when dealing with data that is stored in tabular form. In this case, the index register would be loaded with the starting, or base, address of the table. To find a given element in the table, we would then only need to know its position (i.e., displacement) from the beginning of the table. This would then become the displacement value used in the instruction. Indexed loads require three bytes of code — two for the op code and one more for the displacement. A typical instruction might look like this:

```
DD 46 04  LD B,(IX + 4)
```

EXTENDED ADDRESSING

When passing data to or from the A register, a special form of *extended addressing* can also be used. This is sometimes referred to as *direct addressing* because the address for the memory location is directly contained within the instruction. In fact, it is palced in the two bytes following the op code. As will be the case with all addresses stored by the Z80, the low byte of the address is stored first. For example

```
3A 00 10  LD A, (1000h)
```

With this instruction the contents of memory location 1000h would get loaded into the A register.

IMPLIED ADDRESSING

The last four load operations in Table 10–1 refer to transfers between the A register and the special purpose I and R registers. We won't have too much to say about these instructions except that they define another type of addressing mode. Since both the source and destination locations are completely specified by the op code, these instructions are said to use *implied addressing*. Also note that LD A,I and LD A,R are the only 8-bit load instructions that affect the flag bits: the H and N bits are reset, the S and Z bits are set according to the value loaded, and the P/V flag gets set to the state of a special-purpose circuit within the Z80 called the Interrupt Flip-Flop.

16-BIT LOAD GROUP

Next we shall describe the 16-bit Load group. These instructions perform exactly like their 8-bit counterparts except that 2 bytes get transferred at the same time. Table 10–3 and Table 10–4 outline this group of instructions.

The first thing to notice from Table 10–4 is that almost no register-pair to register-pair loads are allowed. The exception is between the special SP register and the IX and IY registers. Most register pairs can be loaded with an immediate 16-bit value, however. This is known as *immediate extended* addressing. The value is contained in the two bytes following the op code (low byte first, of course). So we could have:

```
21 00 10   LD HL,1000h
```

Table 10–3. The 16-Bit LOAD Group (Courtesy MOSTEK, Corp.)

Mnemonic	Symbolic Operation	Flags								Op-Code		No. of Bytes	No. of M Cycles	No. of T States	Comments
		S	Z		H		P/V	N	C	76 543 210	Hex				
LD dd, nn	dd ← nn	•	•	X	•	X	•	•	•	00 dd0 001 ← n → ← n →		3	3	10	dd Pair 00 BC 01 DE 10 HL 11 SP
LD IX, nn	IX ← nn	•	•	X	•	X	•	•	•	11 011 101 00 100 001 ← n → ← n →	DD 21	4	4	14	
LD IY, nn	IY ← nn	•	•	X	•	X	•	•	•	11 111 101 00 100 001 ← n → ← n →	FD 21	4	4	14	
LD HL, (nn)	H ← (nn+1) L ← (nn)	•	•	X	•	X	•	•	•	00 101 010 ← n → ← n →	2A	3	5	16	
LD dd; (nn)	dd_H ← (nn+1) dd_L ← (nn)	•	•	X	•	X	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	ED	4	6	20	
LD IX, (nn)	IX_H ← (nn+1) IX_L ← (nn)	•	•	X	•	X	•	•	•	11 011 101 00 101 010 ← n → ← n →	DD 2A	4	6	20	
LD IY, (nn)	IY_H ← (nn+1) IY_L ← (nn)	•	•	X	•	X	•	•	•	11 111 101 00 101 010 ← n → ← n →	FD 2A	4	6	20	
LD (nn), HL	(nn+1) ← H (nn) ← L	•	•	X	•	X	•	•	•	00 100 010 ← n → ← n →	22	3	5	16	
LD (nn), dd	(nn+1) ← dd_H (nn) ← dd_L	•	•	X	•	X	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	ED	4	6	20	
LD (nn), IX	(nn+1) ← IX_H (nn) ← IX_L	•	•	X	•	X	•	•	•	11 011 101 00 100 010 ← n → ← n →	DD 22	4	6	20	
LD (nn), IY	(nn+1) ← IY_H (nn) ← IY_L	•	•	X	•	X	•	•	•	11 111 101 00 100 010 ← n → ← n →	FD 22	4	6	20	
LD SP, HL	SP ← HL	•	•	X	•	X	•	•	•	11 111 001	F9	1	1	6	
LD SP, IX	SP ← IX	•	•	X	•	X	•	•	•	11 011 101 11 111 001	DD F9	2	2	10	
LD SP, IY	SP ← IY	•	•	X	•	X	•	•	•	11 111 101 11 111 001	FD F9	2	2	10	
PUSH qq	(SP-2) ← qq_L (SP-1) ← qq_H	•	•	X	•	X	•	•	•	11 qq0 101		1	3	11	qq Pair 00 BC 01 DE 10 HL 11 AF
PUSH IX	(SP-2) ← IX_L (SP-1) ← IX_H	•	•	X	•	X	•	•	•	11 011 101 11 100 101	DD E5	2	4	15	
PUSH IY	(SP-2) ← IY_L (SP-1) ← IY_H	•	•	X	•	X	•	•	•	11 111 101 11 100 101	FD E5	2	4	15	
POP qq	qq_H ← (SP+1) qq_L ← (SP)	•	•	X	•	X	•	•	•	11 qq0 001		1	3	10	
POP IX	IX_H ← (SP+1) IX_L ← (SP)	•	•	X	•	X	•	•	•	11 011 101 11 100 001	DD E1	2	4	14	
POP IY	IY_H ← (SP+1) IY_L ← (SP)	•	•	X	•	X	•	•	•	11 111 101 11 100 001	FD E1	2	4	14	

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
e.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ flag is affected according to the result of the operation.

Table 10–4. Op Codes for 16-Bit LOAD Group (Courtesy MOSTEK, Corp.)

			SOURCE								
			REGISTER						IMM. EXT.	EXT. ADDR.	REG. INDIR.
			AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
DESTINATION	REGISTER	AF										F1
		BC								01 n n	ED 4B n n	C1
		DE								11 n n	ED 5B n n	D1
		HL								21 n n	2A n n	E1
		SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
		IX								DD 21 n n	DD 2A n n	DD E1
		IY								FD 21 n n	FD 2A n n	FD E1
	EXT. ADDR.	(nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
PUSH INSTRUCTIONS →	REG. IND.	(SP)	F5	C5	D5	E5		DD E5	FD E5			

NOTE: The Push & Pop Instructions adjust the SP after every execution

↑ POP INSTRUCTIONS

Of course, this is equivalent to the following 8-bit loads:

```
2E ØØ  LD L,ØØh
26 1Ø  LD H,1Øh
```

As can be seen, the 16-bit instruction uses one less byte and it also executes faster.

We can also use extended or direct addressing to specify a given memory location. But this only gives us a single 8-bit value — not much use for a 16-bit load instruction. Therefore, the CPU must look at another location to get the other half of the 16-bit data. This is exactly what the Z80 does, as shown by the symbolic representation of the LD HL, (nn) instruction. The L register is loaded with the contents of location nn; the H register is loaded with the contents of location nn + 1. This conforms to the Z80's standard of storing the lower order byte of a 16-bit value first. Thus, if memory locations 1000h and 1001h contained an important memory address, we could load the complete 16-bit address with a single command:

```
2A 00 10 LD HL,(1000h)
```

The last form of 16-bit load instruction uses the Stack Pointer (SP) register as a memory pointer (indirect addressing). These instructions are referred to as the PUSH and POP instructions because they operate on the machine stack. What makes these instructions different, aside from using the register indirect addressing mode, is that they also alter the SP register itself after executing. That is, after each PUSH instruction, the SP register is decremented by 2. Conversely each POP instruction adds 2 to the Stack Pointer. Although not explicitly shown in the symbolic descriptions of these operations, you should remember that a PUSH instruction also performs SP ⧫ SP − 2. And POP does an SP ⧫ SP + 2. None of the flag bit are affected by a 16-bit load instruction.

11
Program Flow: JUMP, CALL, and RETURN

Having machine language programs execute in sequential order from start to finish is not very useful. Computer programs (whether they're machine language or BASIC) gain versatility through the use of decision making and branching instructions. Structured programming and memory savings are the benefits derived from the use of *subroutines*. In BASIC, we have the GO TO, GO SUB and RETURN statements. Their machine language equivalents are the JUMP, CALL, and RETURN groups.

Table 11-1 and Table 11-2 outline the JUMP instructions. The first type of jump is called an *unconditional jump* and it has the form:

```
JP nn
```

Like the BASIC GO TO statement, this instruction always causes the program to jump to the address *nn* and then continue executing there. Note the symbolic form for this instruction:

```
PC ← nn
```

This instruction merely places the two-byte address *nn* into the Program Counter (PC) register. Since the PC register determines from where the next instruction will be read, altering the PC has the effect of jumping to a new sequence of program instructions.

The next form of jump instruction is known as the *conditional jump*. This resembles the IF . . . THEN GO TO structure used in BASIC. The conditions which can be tested are the states of several flag bits. These are shown in Table 11-2 under the heading "CONDITION." If the specified condition is met, then the jump instruction will be executed. Otherwise, the program continues with the next sequential instruction. To see an example, suppose we have the following two program lines:

Table 11–1. JUMP Group (Courtesy MOSTEK, Corp.)

Mnemonic	Symbolic Operation	Flags								Op-Code		No. of	No. of M	No. of T	Comments
		S	Z		H		P/V	N	C	76 543 210	Hex	Bytes	Cycles	States	
JP nn	PC - nn	•	•	X	•	X	•	•	•	11 000 011 - n - - n -	C3	3	3	10	
JP cc, nn	If condition cc is true PC - nn, otherwise continue	•	•	X	•	X	•	•	•	11 cc 010 - n - - n -		3	3	10	cc Condition 000 NZ non zero 001 Z zero 010 NC non carry 011 C carry 100 PO parity odd 101 PE parity even 110 P sign positive 111 M sign negative
JR e	PC - PC + e	•	•	X	•	X	•	•	•	00 011 000 - e-2 -	18	2	3	12	
JR C, e	If C = 0, continue	•	•	X	•	X	•	•	•	00 111 000 - e-2 -	38	2	2	7	If condition not met
	If C = 1, PC - PC+e											2	3	12	If condition is met
JR NC, e	If C = 1, continue	•	•	X	•	X	•	•	•	00 110 000 - e-2 -	30	2	2	7	If condition not met
	If C = 0, PC - PC+e											2	3	12	If condition is met
JR Z, e	If Z = 0 continue	•	•	X	•	X	•	•	•	00 101 000 - e-2 -	28	2	2	7	If condition not met
	If Z = 1, PC - PC+e											2	3	12	If condition is met
JR NZ, e	If Z = 1, continue	•	•	X	•	X	•	•	•	00 100 000 - e-2 -	20	2	2	7	If condition not met
	If Z = 0, PC - PC+e											2	3	12	If condition is met
JP (HL)	PC - HL	•	•	X	•	X	•	•	•	11 101 001	E9	1	1	4	
JP (IX)	PC - IX	•	•	X	•	X	•	•	•	11 011 101 11 101 001	DD E9	2	2	8	
JP (IY)	PC - IY	•	•	X	•	X	•	•	•	11 111 101 11 101 001	FD E9	2	2	8	
DJNZ, e	B - B-1 If B = 0, continue	•	•	X	•	X	•	•	•	00 010 000 - e-2 -	10	2	2	8	If B = 0
	If B ≠ 0, PC - PC+e											2	3	13	If B ≠ 0

Notes: e represents the extension in the relative addressing mode.

e is a signed two's complement number in the range <126, 129>

e-2 in the op-code provides an effective address of pc+e as PC is incremented by 2 prior to the addition of e.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

```
ED 5F        LD A, R
C2 ØØ 1Ø     JPNZ, 1ØØØh
```

The first instruction loads the A register with the contents of the R register. After executing this instruction, the Z flag will be set according to the value just transferred. If the value was zero, then the Z flag will be set (=1). Otherwise, it will be reset (=0). According to this condition, the next step will either jump to location 1000h or continue with the program.

RELATIVE ADDRESSING

The previous jump instructions used immediate extended addressing (the target address is part of the instruction). Another form of jump instruction uses *relative addressing*. This means that the new location to jump to is given in terms of an offset (plus or minus so many bytes) from the present location. Instead of supplying the *absolute* location within the instruction, as done previously, the new destination is given *relative* to the current position. The *Jump Relative* instruction has two major advantages. First, it only uses two bytes of memory (one for the op code; another for the relative offset) instead of three. The second advantage of using relative jumps is that they make the machine language program *relocatable* or position independent. This means that a program written to run at location 1000-1020, for example, could be moved byte for byte into location 2000-2020 and still run correctly. We'll explain this by way of example. Suppose we had the following two lines:

```
1ØØØ- ED 5F        LD A,R
1ØØ2- CS ØØ 1Ø     JPNZ 1ØØØh
```

Table 11–2. Op Codes for JUMP Group (Courtesy MOSTEK, Corp.)

			CONDITION									
			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B≠0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	REG. INDIR.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
DECREMENT B, JUMP IF NON ZERO 'DJNZ'	RELATIVE	PC+e										10 e-2

The first instruction reads the contents of the R register into A, setting the Z flag accordingly. If the contents of R was non-zero, then the second instruction causes the program to loop back to the first instruction. This program will continue looping around until the R register reaches 0 at which point the next instruction would be executed. If you're wondering why the R register would change as this program is looping, remember that the R register is a special refresh counter that the Z80 increments automatically after each instruction. Technically, this program has a 50-50 chance of becoming an endless loop, but that's not important here.

You should notice that some more numbers have been added to the machine language listing. This is necessary because we are now dealing with program jumps. This makes it important to know exactly where the program resides in memory. For this purpose, we will again follow the standard format used by assemblers and start each line by giving the memory location for the first byte of code on that line. This will then be followed by the hex code for the instruction and then, finally, by the mnemonic form of the instruction itself. Of course, when writing programs, we start by writing down the mnemonic form first and then filling in the address and machine codes later.

We can also define *labels* that help keep track of what a program is doing. The last example might be written:

```
LOOP  LD A,R
      JPNZ LOOP
```

By using the label LOOP, we can easily tell what this section of machine code is supposed to do. To hand-assemble the preceding code, we only need to know a starting address for the program. If we assume that it starts at 1000h, then we would write:

```
1000- ED 5F        LOOP  LD A,R
                         JPNZ LOOP
```

The machine code for the first instruction is well defined. When we come to the second instruction, we start by looking up the op code for JPNZ which turns out to be C2. Next, we must determine the address which is to be placed in the next two bytes. For this we must look back to where the label LOOP: was defined (in this case, the previous instruction) and then use that address (i.e., 1000h). Thus, we have the complete instruction codes and can write the program as follows:

```
1ØØØ- ED 5F         LOOP  LD A,R
1ØØ2- C2 ØØ 1Ø            JPNZ 1ØØØh
```

Of course, all the computer cares about are the five bytes: E5, 5F, C2, 00 and 10. These tell the Z80 to do the operation which we have asked, namely sit in a loop until the R register equals zero. However, what if these same five bytes were moved to a different location in memory, say starting at address 2000? If we put these five bytes there and then could somehow *disassemble* them back into mnemonic form, this is what we might see:

```
2ØØØ- ED 5F         LD A,R
2ØØ2- C2 ØØ 1Ø      JPNZ 1ØØØh
```

Well, this certainly looks the same! In fact, the instructions themselves have not changed at all. But look at the second instruction. It still wants to jump to location 1000h. Now, however, we do not know what machine language program (if any!) resides at 1000h. In any case, the program no longer *functions* like it was supposed to. So this type of program is *position dependent*.

Consider now a different approach to writing the same program, using the jump relative instruction:

```
1ØØØ ED 5F          LOOP  LD A,R
1ØØ2 2Ø FC                JRNZ LOOP:
```

In hand-assemblying this program, the first line remains the same. The op code for the JRNZ instruction is 20h; so that's pretty easy. Calculating the offset requires a little practice, however. We'll go through the formal procedure first and then look at a useful shortcut.

When using relative addressing, the CPU reads the offset amount located in the second byte of the instruction. This value (it is a signed integer in the range −128 to +127) is then added to the program counter PC register to achieve the address for the next instruction to be executed. However, when the offset is being added to the PC, the PC register has already incremented to point at the address for the instruction following the JR instruction. Therefore, if it is a *conditional* jump relative and the condition is not met, the Program Counter is all prepared to fetch the next sequential instruction.

However, if the condition is met (or if it was an unconditional jump), then the offset is added and the jump performed. If we were willing to talk about jumps relative to "the start of the next instruction," then we would have no difficulty calculating the offset needed for the JR instruction. Unfortunately, it is more common to think in terms of the jump instruction itself. The start of this instruction is two bytes less than the base used for calculating the jump location. Therefore, if we want to use the JR instruction as the base we will have to deduct two from the offset. This is the way they are shown in Table 11-1 and Table 11-2. Remember, also, that whenever jumping backwards, the twos complement notation is used to specify the negative amount. When jumping more than 20 bytes forward or backward, you should probably calculate the offset using the formula:

Offset = Destination instruction − Current JUMP instruction − 2

or, in our example:

```
1ØØØ - 1ØØ2 - 2 = - 4 = FCh
```

For short jumps, however, it is sometimes easier just to count bytes. In the example, this requires counting backwards in hex. If we start at the end of the JR instruction and call that byte FFh, then count back to the start of the previous instruction (the target of the jump), we would call the "20" byte FE, the "5F" byte FD and, finally, the "ED" byte FC. This value, FCh, is, therefore, the correct offset for the JR instruction.

We'll now see what happens if we move this program to a new location. Taking these four bytes (note how the relative addressing also saves one byte) and placing them at location 2000h, we would have:

```
2ØØØ - ED 5F    LD A,R
2ØØØ - 2Ø FC    JPNZ 2ØØØ
```

This program will operate identically to the previous one. Note that relative addressing allows this program to be moved without changing the way it operates. This type of program is called position independent or relocatable.

REGISTER INDIRECT JUMPS

The Z80 also has the ability to jump unconditionally to the address pointed to by one of the registers HL, IX, or IY. Thus any instructions which affect the contents of these registers can be used to set up the address for a jump.

AUTOMATIC LOOPING

The last instruction in the jump group is rather special. It is the Decrement B, Jump if Non Zero or DJNZ instruction. This is sort of like a FOR . . . NEXT loop in BASIC, with *B* as the control variable. To use this instruction, the B register is first loaded with the number of times that the loop is to be executed (up to 255). Then the desired code to be repeated is written. This is followed by the DJNZ instruction which points back to the beginning of the loop. The following example will load memory location 2000h with every possible value one at a time.

```
1000 - 21 00 20          LD HL,2000h
1003 - 06 00             LB B,0
1005 - 70           LOOP LD (HL),B
1006 - 10 FD             DJNZ LOOP
```

Let's analyze this program in detail. The first line sets up the HL register pair to point to location 2000h. Line two initializes the B register with a value of zero. As we shall see, this causes the loop to be executed 256 times. The third line begins the loop and stores the contents of register B into the memory location pointed to by HL. Therefore, memory location 2000h now holds a value of zero.

The last line of the program does two things. First, it decrements B and then jumps back to LOOP: if B is nonzero. On the first pass through this instruction, the B register equals zero. Therefore, decrementing B gives it a value of −1 or FFh. If you prefer, you can also think of this as the number 255. In either case, B certainly is not zero so the jump back to LOOP: is executed. Now, the value FFh gets stored at location 2000h, B decrements to FEh, and the loop repeats again. This continues for 255 more times, decrementing the contents of 2000h by one for each pass through the loop.

Finally, after storing the value of 01 at 2000h, register B is decremented, leaving a value of zero. At this point, the conditional jump is not satisfied and, therefore, the program continues with the next sequential instruction.

CALL AND RETURN GROUP

In much the same way in which the GO SUB and RETURN statements are used in BASIC, machine language programmers can use the CALL and RETurn instructions. Table 11–3 and Table 11–4 show the various forms of these instructions.

Table 11–3. CALL and RETURN Group (Courtesy MOSTEK, Corp.)

Mnemonic	Symbolic Operation	Flags								Op-Code				No. of Bytes	No. of M Cycles	No. of T States	Comments
		S	Z		H		P/V	N	C	76	543	210	Hex				
CALL nn	$(SP-1) \leftarrow PC_H$ $(SP-2) \leftarrow PC_L$ $PC \leftarrow nn$	•	•	X	•	X	•	•	•	11 ← ←	001 n n	101 → →	CD	3	5	17	
CALL cc, nn	If condition cc is false continue, otherwise same as CALL nn	•	•	X	•	X	•	•	•	11 ← ←	cc n n	100 → →		3 3	3 5	10 17	If cc is false If cc is true
RET	$PC_L \leftarrow (SP)$ $PC_H \leftarrow (SP+1)$	•	•	X	•	X	•	•	•	11	001	001	C9	1	3	10	
RET cc	If condition cc is false continue, otherwise same as RET	•	•	X	•	X	•	•	•	11	cc	000		1 1	1 3	5 11	If cc is false If cc is true
RETI	Returr from interrupt	•	•	X	•	X	•	•	•	11 01	101 001	101 101	ED 4D	2	4	14	
RETN[1]	Return from non maskable interrupt	•	•	X	•	X	•	•	•	11 01	101 000	101 101	ED 45	2	4	14	
RST p	$(SP-1) \leftarrow PC_H$ $(SP-2) \leftarrow PC_L$ $PC_H \leftarrow 0$ $PC_L \leftarrow p$	•	•	X	•	X	•	•	•	11	t	111		1	3	11	

cc	Condition	
000	NZ	non zero
001	Z	zero
010	NC	non carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	p
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

[1] RETN loads $IFF_2 \leftarrow IFF_1$

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

Table 11–4. Op Codes for CALL and RETURN Group (Courtesy MOSTEK, Corp.)

			CONDITION									
			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B≠0
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REG. INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REG. INDIR.	(SP) (SP+1)	ED 45									

NOTE—CERTAIN FLAGS HAVE MORE THAN ONE PURPOSE. REFER TO SECTION 6.0 FOR DETAILS

		OP CODE	
CALL ADDRESS	0000_H	C7	'RST 0'
	0008_H	CF	'RST 8'
	0010_H	D7	'RST 16'
	0018_H	DF	'RST 24'
	0020_H	E7	'RST 32'
	0028_H	EF	'RST 40'
	0030_H	F7	'RST 48'
	0038_H	FF	'RST 56'

When writing a BASIC statement such as GO SUB 5000, it is quite obvious where the program should continue executing. The line number, 5000, is right there in the instruction. Have you ever wondered, however, how the computer knows where to go when it reaches a RETURN statement, especially when one subroutine may have called another subroutine (i.e., they may be nested)? The answer is quite simple. Whenever a GO SUB statement is executed, the computer keeps track of the line number it is currently on in an area of memory called the *GO SUB Stack*. Like all LIFO (Last in — First OUT) stacks the line number of the most recently executed GO SUB is always kept on the top.

A GO SUB statement, therefore, requires that the current line number be PUSHED onto this stack. The statement number is also saved, since multiple statements can be placed on one line. Whenever a RETURN statement is encountered, the computer simply POPs the line and statement numbers off the GO SUB stack and continues executing with the next statement just after the GO SUB.

The machine language CALL instruction does much the same thing. It uses the machine stack to store the current address (two bytes) and then loads the Program Counter with the target address for the jump. You can also see this by looking at the symbolic description shown for the CALL nn instruction. As with other instructions that use the SP register, there is also the hidden operation of updating the contents of the SP register. You can add SP ← SP-2 to the description of the CALL instruction if you want to be technically complete. Notice that there is also a *conditional call* instruction that can first test one of eight flag conditions.

If you've followed everything so far, then it should be quite obvious how the RETurn instruction works. It simply POPs two bytes off the stack and stores them in the program counter. There is one big *caution* that needs to be mentioned here. Unlike the BASIC intepreter, which keeps a separate stack for pending GO SUB instructions, the machine language CALL uses the Z80 machine stack. This stack may also be used by other parts of a program temporarily to store data, addresses, status, etc. Therefore, it is imperative that the stack be in the proper state when the RETurn instruction is executed. The RETurn instruction can also be made conditional on the status of the flag bits.

There are two more RETurn instructions that are used when the interrupt facilities of the Z80 are employed. These instructions perform the same operation as a RETurn, but have special features that make them specifically applicable to interrupt routines. The RETI instruction is used to return from an IRQ service routine and the RETN instruction should be used to end a Nonmaskable Interrupt routine.

RESTART GROUP

The final group of instructions in this category is the Restart instructions. These are eight, special-purpose instructions that *CALL* specific locations in page zero of memory. The advantages of these instructions are that they only require one byte and they execute much faster than the normal CALL instruction. Routines that are used over and over are good candidates for use by the restart instructions.

12
Arithmetic and Logic Operations

The Z80 instructions discussed so far have only dealt with one or two bytes of information. The load instructions, for example, simply move data from one location to another. The instructions we are now going to describe operate on two pieces of information in some defined manner to create a third. Thus we will have two source locations, as well as a destination register, to hold the result. To keep things simple, most of these instructions use the special-purpose A register or accumulator to hold one of the source operands and also to hold the result of the operation. All of these instructions affect the flag register according to the results of the operation. Refer to Table 12–1 and Table 12–2 as we describe the arithmetic and logic operations of the Z80.

Z80 INSTRUCTIONS

ADD, ADC

There are two types of Z80 instructions that perform addition. The first instruction, with the mnemonic ADD, simply sums the contents of the accumulator with a specified 8-bit value from another location. The result is placed in the accumulator and the flag register is set accordingly. The second operand can come from any of the other registers or from a memory location specified either directly or via the index registers.

The second instruction, ADd with Carry (ADC), is identical to the ADD instruction except that the value of the Carry flag (1 or 0) is added in, also. This makes it possible to do multiple byte, or multiple precision addition. Suppose, for example, we wanted to add two 16-bit, unsigned integers. Assuming we could only work with eight bits at a

Table 12–1. 8-Bit ARITHMETIC and LOGICAL Group (Courtesy MOSTEK, Corp.)

Mnemonic	Symbolic Operation	Flags								Op-Code		No. of Bytes	No. of M Cycles	No. of T States	Comments
		S	Z		H		P/V	N	C	76 543 210	Hex				
ADD A, r	A ← A + r	↕	↕	X	↕	X	V	0	↕	10 $\boxed{000}$ r		1	1	4	r Reg.
ADD A, n	A ← A + n	↕	↕	X	↕	X	V	0	↕	11 $\boxed{000}$ 110 ← n →		2	2	7	000 B 001 C 010 D
ADD A, (HL)	A ← A+(HL)	↕	↕	X	↕	X	V	0	↕	10 $\boxed{000}$ 110		1	2	7	011 E
ADD A, (IX+d)	A ← A+(IX+d)	↕	↕	X	↕	X	V	0	↕	11 011 101 10 $\boxed{000}$ 110 ← d →	DD	3	5	19	100 H 101 L 111 A
ADD A, (IY+d)	A ← A+(IY+d)	↕	↕	X	↕	X	V	0	↕	11 111 101 10 $\boxed{000}$ 110 ← d →	FD	3	5	19	
ADC A, s	A ← A+s+CY	↕	↕	X	↕	X	V	0	↕	$\boxed{001}$					s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction. The indicated bits replace the $\boxed{000}$ in the ADD set above.
SUB s	A ← A - s	↕	↕	X	↕	X	V	1	↕	$\boxed{010}$					
SBC A, s	A ← A - s - CY	↕	↕	X	↕	X	V	1	↕	$\boxed{011}$					
AND s	A ← A ∧ s	↕	↕	X	1	X	P	0	0	$\boxed{100}$					
OR s	A ← A ∨ s	↕	↕	X	0	X	P	0	0	$\boxed{110}$					
XOR s	A ← A ⊕ s	↕	↕	X	0	X	P	0	0	$\boxed{101}$					
CP s	A - s	↕	↕	X	↕	X	V	1	↕	$\boxed{111}$					
INC r	r ← r + 1	↕	↕	X	↕	X	V	0	•	00 r $\boxed{100}$		1	1	4	
INC (HL)	(HL) ← (HL)+1	↕	↕	X	↕	X	V	0	•	00 110 $\boxed{100}$		1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	↕	↕	X	↕	X	V	0	•	11 011 101 00 110 $\boxed{100}$ ← d →	DD	3	6	23	
INC (IY+d)	(IY+d) ← (IY+d)+1	↕	↕	X	↕	X	V	0	•	11 111 101 00 110 $\boxed{100}$ ← d →	FD	3	6	23	
DEC s	s ← s - 1	↕	↕	X	↕	X	V	1	•	$\boxed{101}$					s is any of r, (HL), (IX+d), (IY+d) as shown for INC. DEC same format and states as INC. Replace $\boxed{100}$ with $\boxed{101}$ in OP Code.

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow, P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

time, we would start by adding the lower order bytes of the two numbers together. If this addition caused a carry-over into the next higher place, then we would have to add in the carry when we perform the addition on the higher order bytes. See Fig. 12–1.

Writing a program to accomplish this procedure is relatively simple. First we decide where the operands will come from and where to store the 16-bit result. As a quick example, we'll write a program to add the

Table 12–2. Op Codes for 8-Bit ARITHMETIC and LOGIC Group (Courtesy MOSTEK, Corp.)

	SOURCE										
	REGISTER ADDRESSING							REG. INDIR.	INDEXED		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

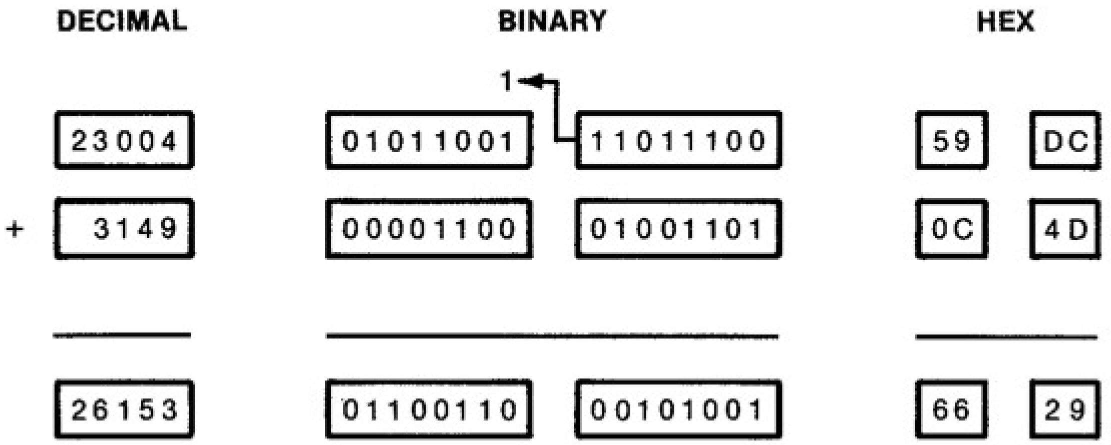


Fig. 12–1. Addition, with carry, of two 16-bit numbers.

Listing 12–1

```
7ØØØ-11 DC 59     LD DE,59DCh       17,22Ø,89
7ØØ3- 21 4D ØC    LD HL,ØC4Dh       33,77,12
7ØØ6- 7B          LD A, E           123
7ØØ7- 85          ADD A, L          133
7ØØ8- 4F          LD C, A           79
7ØØ9- 7A          LD A, D           122
7ØØA- 8C          ADC A, H          14Ø
7ØØB- 47          LD B, A           71
7ØØC- C9          RET               2Ø1
```

Listing 12–2

```
 1Ø   FOR a=28672 TO 28684
 2Ø   READ d: POKE a,d
 3Ø   NEXT a
 4Ø   DATA 17,22Ø,89,33,77,12,123,133,79,122,14Ø,71,2Ø1
 9Ø   : REM Now let's do it!
1ØØ   PRINT USR 28672
```

contents of the DE register to the HL register, storing the results in BC. Listing 12–1 shows the result.

The program starts by loading the two 16-bit operands into the DE and HL registers. To perform the addition of the lower bytes, we first load one operand into the accumulator. In this case, we pick the E register (i.e., the lower half of the DE pair). Next, we add the other low byte, in the L register, to the accumulator. This gives the lower order byte of the result which is then loaded into register C. At this point, if there was a carry from the first addition, the carry flag will be set. Otherwise, it will be reset. With the sample numbers chosen, it will be set because there is a carry. The accumulator is next loaded with one of the high bytes — the D register — in preparation for the second addition.

The next instruction is an Add with Carry of the two high bytes (D, which already is in A, plus H). This allows any carry over from the previous addition also to get added in. After loading the high byte of the answer into register B, the program RETurns to BASIC. Note that when running this program from BASIC, we can easily print out the answer stored in the BC register by using the PRINT USR call.

Listing 12–2 shows how to enter and RUN this program on the T/S 2068. You should verify that the result printed is, indeed, the correct answer for the two numbers used. In fact, by changing the second/

third and fourth/fifth bytes in the DATA statement, you can make this program add any two numbers.

SUB, SBC

Everything we have said about the addition instructions ADD and ADC, applies to the subtraction instructions SUB and SBC (SUBtract and SuBtract with Carry). The only difference, aside from the opposite operation being performed, is that the Carry flag is now used to represent a *borrow* condition. Listing 12–3 shows an example of 8-bit subtraction. This time, both the operands and the result are stored in memory locations. We can, therefore, POKE the minuend and subtrahend into memory, call the subtraction routine with RANDOMIZE USR, and then PEEK at the remainder. Listing 12–4 shows how to do this from BASIC.

Remember that this is a very simple routine. It does not check for any borrow condition and only handles 8-bit quantities. Of course, we can always write a multiple precision subtraction routine as we did for addition.

Listing 12–3

```
7ØØØ- 3A ØB 7Ø   LD A, (7ØØBh)      58,11,112
7ØØ3- 21 ØC 7Ø   LD HL, (7ØØCh)     33,12,112
7ØØ6- 96         SUB A,(HL)         15Ø
7ØØ7- 32 ØD 7Ø   LD (7ØØDh),A       5Ø,13,112
7ØØA- C9         RET                2Ø1

7ØØB  : Contains Minuend
7ØØC  : Contains Subtrahend
7ØØD  : Receives Remainder
```

Listing 12–4

```
  1Ø   FOR a=28672 TO 28682
  2Ø   READ d: POKE a,d
  3Ø   NEXT a
  4Ø   DATA 58,11,112,33,12,112,15
Ø,5Ø ,  13,112,2Ø1
  6Ø   : REM Store minuend
  7Ø   POKE 28683,15Ø
  8Ø   : REM and subtrahend
  9Ø   POKE 28684,35
 1ØØ   RANDOMIZE USR 28672
 19Ø   : REM Now print the result
 2ØØ   PRINT PEEK 28685
```

SIGNED ARITHMETIC

So far, we have discussed addition and subtraction of whole, positive integers. Even our multiple precision addition routine is based on these assumptions. We'll now see what happens if we try to apply these routines to *signed* integers.

Suppose we wish to add 100 + 75, using signed arithmetic. The problem would look something like this:

$$\begin{array}{rr} (\text{1ØØ}_{10}) & \text{Ø11ØØ1ØØ} \\ +\ (75_{10}) & \text{Ø1ØØ1Ø11} \\ \hline & \text{C} \qquad\qquad\ \\ (-81_{10}) & \text{Ø}\ \ \text{1Ø1Ø1111} \end{array}$$

We know the answer is not −81, so something must have gone wrong. The carry flag is not set, so the error is not due to a carry past the eighth bit. What has happened, however, is that the true answer, 175, has exceeded the maximum amount that can be represented in twos complement form by eight bits (i.e., +127). This is known as an *overflow* condition. Likewise, whenever an operation results in a value less than −128, we would have an *underflow* condition. In either case, the accumulator contains an erroneous value which can create all kinds of havoc, if it goes unnoticed.

Fortunately, the Z80 knows about signed numbers, so it has a built-in overflow/underflow detector which appears to the programmer as the P/V bit in the flag register. Actually, the P/V (Parity/oVerflow) flag serves two purposes, depending upon whether the Z80 is performing a logical or arithmetic operation. For now, we are interested in its use as an overflow (or underflow) flag.

Whenever an arithmetic operation, such as ADD or SUB is performed, the P/V flag is set according to the result of the operation. If the resulting value is between −128 and +127, and, therefore, is a valid signed integer, then the P/V flag will be cleared (i.e., set to 0); otherwise, it will be set. By testing this flag or using a conditional jump, the program can be diverted to handle the overflow condition properly.

CP

The Z80 has a special instruction for ComParing the binary values of the accumulator with any other location. What makes this instruction special is that, unlike all of othe other arithmetic operations, no result gets stored. Actually, the CP instruction is just like the SUB instruction except that the accumulator is not changed. The flag register is affected, however, so this provides a way of comparing two values. This instruction is usually followed by a conditional jump such as JP Z (jump if zero, implying that the two values were equal) or JP NZ (jump if nonzero, implying that the two values were unequal). Other instructions, such as JP C (carry, indicating that the accumulator was less than the other value) can also be used. Since the accumulator is not affected by the CP instruction, a series of comparisons can be made one after the other until a match is finally made. For example:

```
LD          A, DATAVALUE
CP          Ø1
JP Z        ROUTINE1
CP          Ø2
JP Z        ROUTINE2
CP          Ø3
JP Z        ROUTINE3
JP          NONE OF THE ABOVE
```

INC, DEC

The last two arithmetic functions supported by the Z80 involve incrementing (adding 1) and decrementing (subtracting 1) registers or memory locations. They do not require use of the accumulator, since only one operand is required. (The other operand is *implied* as 1 and the destination for the result is the same as the source.) The INC and DEC instructions can be applied to any register or any memory location via indirect (HL) or indexed (IX+d, IY+d) addressing.

Logical Operations — AND, OR, XOR, NOT

Aside from the arithmetic operations just described, the Z80 can also perform four basic logic functions on data values. Three of these require two operands — AND, OR, and XOR — and will be described here. The fourth function, NOT or complement, only requires one byte and, in fact, can *only* operate on the accumulator. It will be described in a later group of instructions.

Performing the AND, OR and eXclusive OR instructions is very similar to the arithmetic functions. One operand is held in the accumulator, as is the final result. The other operand can come from any register or memory location. When the two operands are brought together inside the Z80's ALU, the appropriate logic function is then applied to each pair of bits. Each bit in the result is determined by the logic function and the values of the same bit position in each operand. The following examples should make this clear:

```
      10011010          10011010          10011010
AND               OR                XOR
      01011110          01011110          01011110

    = 00011010        = 11011110        = 11000100
```

When performing logical operations, the P/V flag is used to denote the *parity* of the result in the accumulator. Parity simply refers to number of 1 bits in the answer. If there are an even number, then the result is said to have *even parity*. An odd number means *odd parity*. For example, the binary number 10111011 has six 1 bits and, therefore, has even parity. The P/V flag is set when the parity of the result is even; it is reset if the parity is odd.

The parity of a number can serve many purposes. In ASCII representations, for example, which use a 7-bit code, the parity value can be placed in the eighth bit and sent along with the character. When transmitting data over long distances (e.g., through the telephone lines), the parity information can be used to detect any errors in transmission. If one bit of the code is received incorrectly, then the parity of the data at the receiving end will not match that which was sent. This *parity checking* can alert the receiving device that it has incorrect data. Of course, if more than one bit gets changed during transmission, this scheme will not always work.

16-BIT ARITHMETIC GROUP

Refer to Table 12–3 and Table 12–4 for information on the 16-bit arithmetic instructions. In general, the 16-bit operations are identical to their 8-bit counterparts. Watch out for the flag bits, however, as each instruction only affects certain flags. You might also like to note that the

Table 12–3. 16-Bit ARITHMETIC and LOGICAL Group (Courtesy MOSTEK, Corp.)

	Symbolic	Flags								Op-Code		No. of	No.of M	No.of T		
Mnemonic	Operation	S	Z		H		P/V	N	C	76 543 210	Hex	Bytes	Cycles	States	Comments	
ADD HL, ss	HL ← HL+ss	•	•	X	X	X	•	0	↕	00 ss1 001		1	3	11	ss	Reg.
															00	BC
ADC HL, ss	HL ← HL+ss+CY	↕	↕	X	X	X	V	0	↕	11 101 101	ED	2	4	15	01	DE
										01 ss1 010					10	HL
															11	SP
SBC HL, ss	HL ← HL-ss-CY	↕	↕	X	X	X	V	1	↕	11 101 101	ED	2	4	15		
										01 ss0 010						
ADD IX, pp	IX ← IX + pp	•	•	X	X	X	•	0	↕	11 011 101	DD	2	4	15	pp	Reg.
										00 pp1 001					00	BC
															01	DE
															10	IX
															11	SP
ADD IY, rr	IY ← IY + rr	•	•	X	X	X	•	0	↕	11 111 101	FD	2	4	15	rr	Reg.
										00 rr1 001					00	BC
															01	DE
															10	IY
															11	SP
INC ss	ss ← ss + 1	•	•	X	•	X	•	•	•	00 ss0 011		1	1	6		
INC IX	IX ← IX + 1	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	10		
										00 100 011	23					
INC IY	IY ← IY + 1	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	10		
										00 100 011	23					
DEC ss	ss ← ss - 1	•	•	X	•	X	•	•	•	00 ss1 011		1	1	6		
DEC IX	IX ← IX - 1	•	•	X	•	X	•	•	•	11 011 101	DD	2	2	10		
										00 101 011	2B					
DEC IY	IY ← IY - 1	•	•	X	•	X	•	•	•	11 111 101	FD	2	2	10		
										00 101 011	2B					

Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

16-bit addition routine in Fig. 12–2 can be greatly simplified using the 16-bit ADD instruction. See how Listing 12–5 compares to the earlier routine.

BIT OPERATIONS

Some of the most powerful instructions available on the Z80 are the *bit operations*. These instructions allow the programmer to change or test the state of a single bit within any register or memory location. Table 12–5 and Table 12–6 show the various forms of the bit SET, RESet, and TEST instructions.

The SET and RESet instructions affect only the indicated bit within the data location. The flag register is unchanged by these operations.

Table 12–4. Op Codes for 16-Bit ARITHMETIC and LOGICAL Group (Courtesy MOSTEK, Corp.)

			SOURCE					
			BC	DE	HL	SP	IX	IY
DESTINATION	'ADD'	HL	09	19	29	39		
		IX	DD 09	DD 19		DD 39	DD 29	
		IY	FD 09	FD 19		FD 39		FD 29
	ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
	SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
	INCREMENT 'INC.		03	13	23	33	DD 23	FD 23
	DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

Listing 12–5

```
7ØØØ-  3A  ØB  7Ø   LD A, (7ØØBh)      58,11,112
7ØØ3-  21  ØC  7Ø   LD HL, (7ØØCh)     33,12,112
7ØØ6-  19           ADD HL,DE          25
7ØØ7-  44           LD B,H             68
7ØØ8-  4D           LD C,L             77
7ØØ9-  C9           RET                2Ø1
```

The BIT instruction examines one bit of the desired location and then sets the Z flag according to that value. Note that if the bit is a zero, the Zero flag is set (=1). Therefore, the Z flag actually represents the complement of the bit tested.

Bit operations serve many purposes in machine language programs. They make it easy for a programmer to create external flag registers in memory. These registers can then be used to monitor and control various procedures within a program just as the flag register in the Z80 is used. Bit testing can also be used to tell if a number is odd or even, to check the status of an external device read through an I/O port, or to implement special data structures for efficient memory usage.

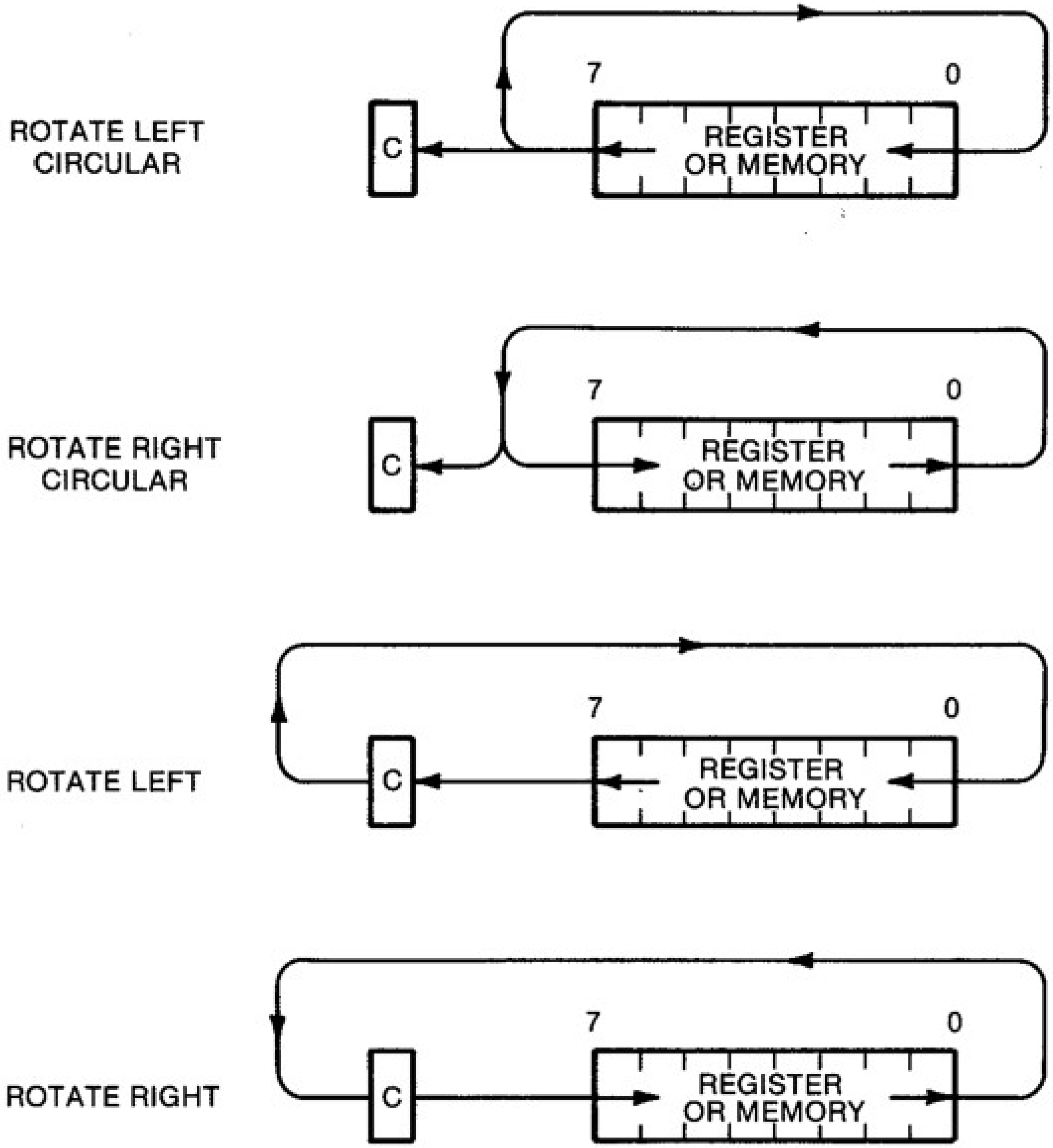


Fig. 12–2. The ROTATE operations.

ROTATE AND SHIFT GROUP

Refer to Table 12–7 and Table 12–8 as we discuss the rotate and shift operations performed by the Z80. These operations generally involve a single eight-bit value plus the carry flag. A *rotate* operation causes each bit in the data location to move one position, either to the right or left. One bit gets rotated out of the operand and into the C flag. Sometimes this same bit is shifted into the opposite end of the operand. This is the case for the eight-bit Rotate Left Circular (RLC) and Rotate Right Circular (RRC) instructions. It is also possible to include the carry flag within the rotation, making for a nine-bit Rotate Left (RL) or Rotate Right (RR). Fig. 12–2 outlines the effect of these four instructions. Note that most of these instructions affect the entire

Table 12–5. SET, RESet, and TEST Group (Courtesy MOSTEK, Corp.)

Mnemonic	Symbolic Operation	Flags								Op-Code		No. of Bytes	No. of M Cycles	No. of T States	Comments	
		S	Z		H		P/V	N	C	76 543 210	Hex					
BIT b, r	$Z \leftarrow \overline{r}_b$	X	↕	X	1	X	X	0	•	11 001 011	CB	2	2	8	r	Reg.
										01 b r					000	B
BIT b, (HL)	$Z \leftarrow \overline{(HL)}_b$	X	↕	X	1	X	X	0	•	11 001 011	CB	2	3	12	001	C
										01 b 110					010	D
BIT b, (IX+d)$_b$	$Z \leftarrow \overline{(IX+d)}_b$	X	↕	X	1	X	X	0	•	11 011 101	DD	4	5	20	011	E
										11 001 011	CB				100	H
										– d –					101	L
										01 b 110					111	A
															b	Bit Tested
BIT b, (IY+d)$_b$	$Z \leftarrow \overline{(IY+d)}_b$	X	↕	X	1	X	X	0	•	11 111 101	FD	4	5	20	000	0
										11 001 011	CB				001	1
										– d –					010	2
										01 b 110					011	3
															100	4
															101	5
															110	6
															111	7
SET b, r	$r_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011	CB	2	2	8		
										$\boxed{11}$ b r						
SET b, (HL)	$(HL)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011	CB	2	4	15		
										$\boxed{11}$ b 110						
SET b, (IX+d)	$(IX+d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 011 101	DD	4	6	23		
										11 001 011	CB					
										– d –						
										$\boxed{11}$ b 110						
SET b, (IY+d)	$(IY+d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 111 101	FD	4	6	23		
										11 001 011	CB					
										– d –						
										$\boxed{11}$ b 110						
RES b, s	$s_b \leftarrow 0$ $s \equiv r, (HL),$ $(IX+d),$ $(IY+d)$	•	•	X	•	X	•	•	•	$\boxed{10}$					To form new Op-Code replace $\boxed{11}$ of SET b, s with $\boxed{10}$. Flags and time states for SET instruction	

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

flag register. Fig. 12–3 shows how the operations would apply to a sample data byte.

Shift operations are very similar to rotates except that the link from the C flag into the data location is broken. Thus, the bit shifted out of the C flag is lost forever and the vacant bit position in the data byte is filled in with either a zero (logical shift) or a repeat of its last value (arithmetic shift). Shifting operations are mostly used to perform binary multiplication and division. As we have shown in Chapter 2, shifting a binary number one bit position to the left is equivalent to

Table 12–6. Op Codes for SET, RESet, and TEST Group (Courtesy MOSTEK, Corp.)

		REGISTER ADDRESSING							REG. INDIR.	INDEXED	
	BIT	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
TEST 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	GB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET BIT 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6

Table 12–6 — cont.Op Codes for SET, RESet, and TEST Group

SET BIT 'SET'	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

multiplying the number by two. Likewise, a right shift implements a division by two.

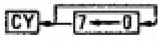
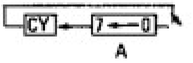
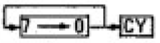
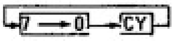
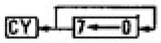
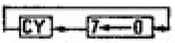
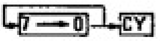
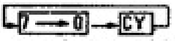
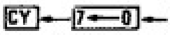
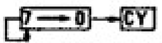
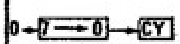
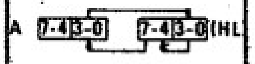
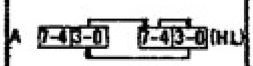
Shifting a data byte always causes one bit to be shifted out into the carry flag. At the other end of the byte, there is an empty bit position which must be filled. Normally, we will want to set this bit equal to zero so that multiplication and division operations give the correct results. The Shift Left Arithmetic (SLA) and Shift Right Logical (SRL) instructions operate in this manner.

The Shift Right Arithmetic (SRA) instruction is used to divide *signed* integers. Since the most significant bit represents the sign of such numbers, filling this position with a zero might change its sign. Therefore, the highest order bit is retained in its position as well as being duplicated into the next position. Fig. 12–4 and Fig. 12–5 outline the results of the various shift operations including the use of the SRA instruction to divide signed numbers.

BCD Rotates

The final two rotate instructions are used on numbers stored in BCD format. These instructions can rotate an entire digit (4 bits) to or from the accumulator. The other two digits involved in the rotate are stored in a memory location which is pointed to by the HL register.

Table 12–7. ROTATE and SHIFT Group (Courtesy MOSTEK, Corp.)

	Symbolic	Flags								Op-Code		No. of	No. of	No. of	
Mnemonic	Operation	S	Z		H		P/V	N	C	76 543 210	Hex	Bytes	M Cycles	T States	Comments
RLCA	 A	•	•	X	0	X	•	0	↕	00 000 111	07	1	1	4	Rotate left circular accumulator
RLA	 A	•	•	X	0	X	•	0	↕	00 010 111	17	1	1	4	Rotate left accumulator
RRCA	 A	•	•	X	0	X	•	0	↕	00 001 111	0F	1	1	4	Rotate right circular accumulator
RRA	 A	•	•	X	0	X	•	0	↕	00 011 111	1F	1	1	4	Rotate right accumulator
RLC r		↕	↕	X	0	X	P	0	↕	11 001 011 00 $\boxed{000}$ r	CB	2	2	8	Rotate left circular register r
RLC (HL)		↕	↕	X	0	X	P	0	↕	11 001 011 00 $\boxed{000}$ 110	CB	2	4	15	r Reg. 000 B 001 C
RLC (IX+d)	 r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	11 011 101 11 001 011 – d – 00 $\boxed{000}$ 110	DD CB	4	6	23	010 D 011 E 100 H 101 L 111 A
RLC (IY+d)		↕	↕	X	0	X	P	0	↕	11 111 101 11 001 011 – d – 00 $\boxed{000}$ 110	FD CB	4	6	23	
RL s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	$\boxed{010}$					Instruction format and states are as shown for RLC's. To form new Op-Code replace $\boxed{000}$ of RLC's with shown code
RRC s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	$\boxed{001}$					
RR s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	$\boxed{011}$					
SLA s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	$\boxed{100}$					
SRA s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	$\boxed{101}$					
SRL s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	$\boxed{111}$					
RLD	 A (HL)	↕	↕	X	0	X	P	0	•	11 101 101 01 101 111	ED 6F	2	5	18	Rotate digit left and right between the accumulator and location (HL). The content of the upper half of the accumulator is unaffected
RRD	 A (HL)	↕	↕	X	0	X	P	0	•	11 101 101 01 100 111	ED 67	2	5	18	

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

MULTIPLICATION AND DIVISION

As we saw in Chapter 2, binary multiplication involves nothing more than three simple steps: bit testing, bit shifting, and addition. Having

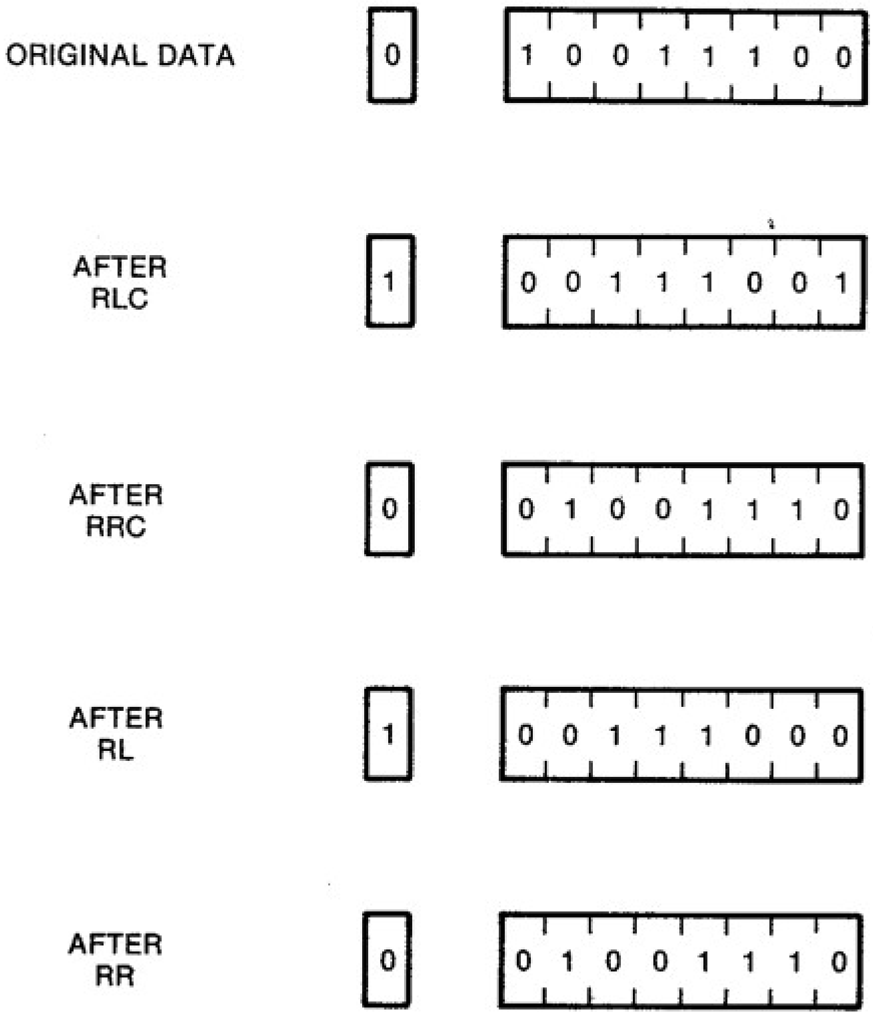


Fig. 12–3. Typical results of the ROTATE operations.

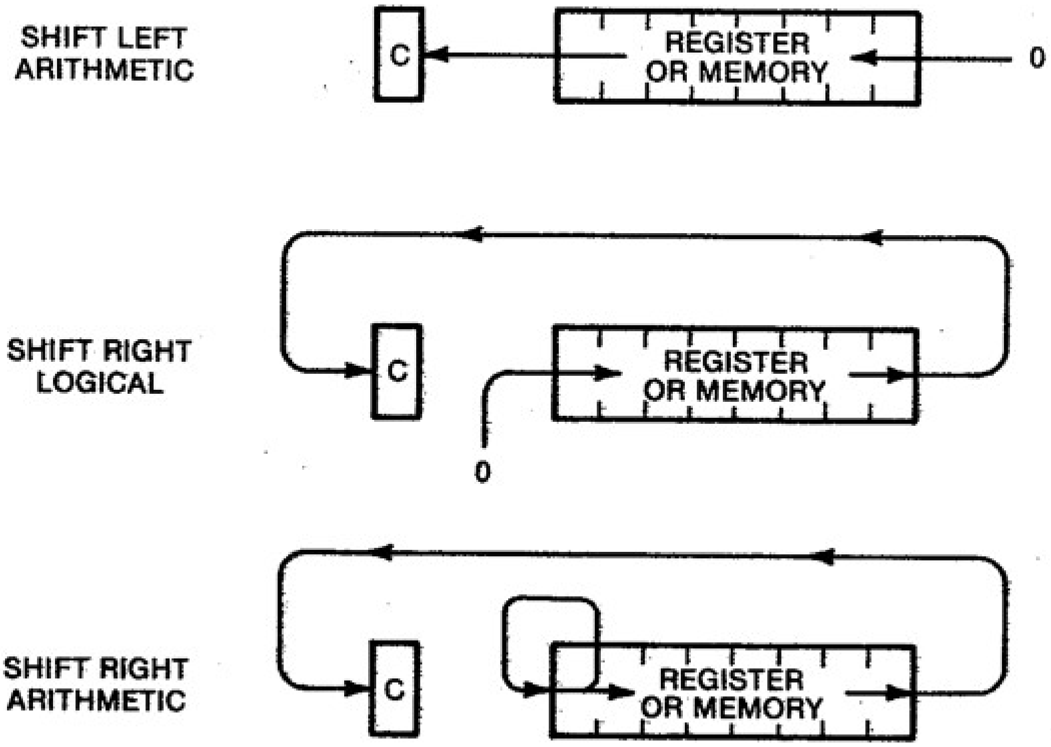


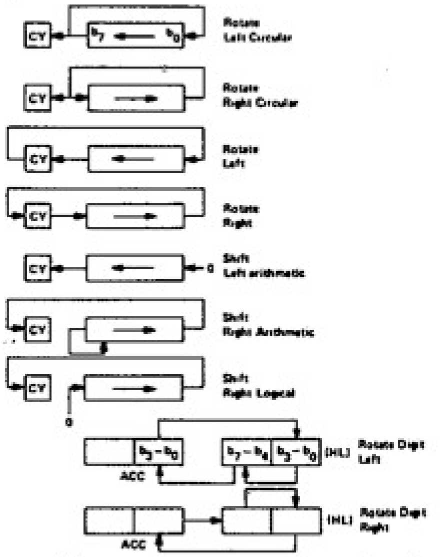
Fig. 12–4. The SHIFT operations.

Table 12–8. Op Codes for ROTATE and SHIFT Group (Courtesy MOSTEK, Corp.)

Source and Destination

TYPE OF ROTATE OR SHIFT		A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)
	'RLC'	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB d 06	FD CB d 06
	'RRC'	CB 0F	CB 08	CB 09	CB 0A	CB 0B	CB 0C	CB 0D	CB 0E	DD CB d 0E	FD CB d 0E
	'RL'	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB d 16	FD CB d 16
	'RR'	CB 1F	CB 18	CB 19	CB 1A	CB 1B	CB 1C	CB 1D	CB 1E	DD CB d 1E	FD CB d 1E
	'SLA'	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB d 26	FD CB d 26
	'SRA'	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB d 2E	FD CB d 2E
	'SRL'	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB d 3E	FD CB d 3E
	'RLD'								ED 6F		
	'RRD'								ED 67		

	A
RLCA	07
RRCA	0F
RLA	17
RRA	1F



seen how the Z80 performs all of these operations, we are ready to write some basic multiplication (and division) routines. One thing that you learn very quickly when programming computers — whether in assembly language or BASIC — is that there are always many different ways to write a program. There is seldom ever one way which is, without question, the best. Most of the time, there are *three* "best" ways.

One way is the *shortest*, using the least number of instructions and usually the smallest amount of bytes. This is always the preferred way when you have only a small amount of memory (RAM or ROM) in which to write the program. Another way is the *fastest*, using those instructions with fast execution times. Programmers often resort to some very esoteric tricks to achieve these goals.

The final way to write programs is the one in which the program flow is *easiest to follow*. This method creates programs that are easy to explain and easy for the beginner to understand. The routines we are about to describe were not written for optimum speed or memory usage. But while they may not be the most efficient for a given task, they are still very useful, general purpose routines. These routines also provide an opportune time to add the final item to our assembly language listings: comments. Just like the REMark statement in BASIC, we can add comments to our program listings to make them easier to understand. We do this by adding a semicolon after the last operand in any instruction, or at the beginning of a line. Everything after the semicolon would be ignored by an assembler so we'll do the same.

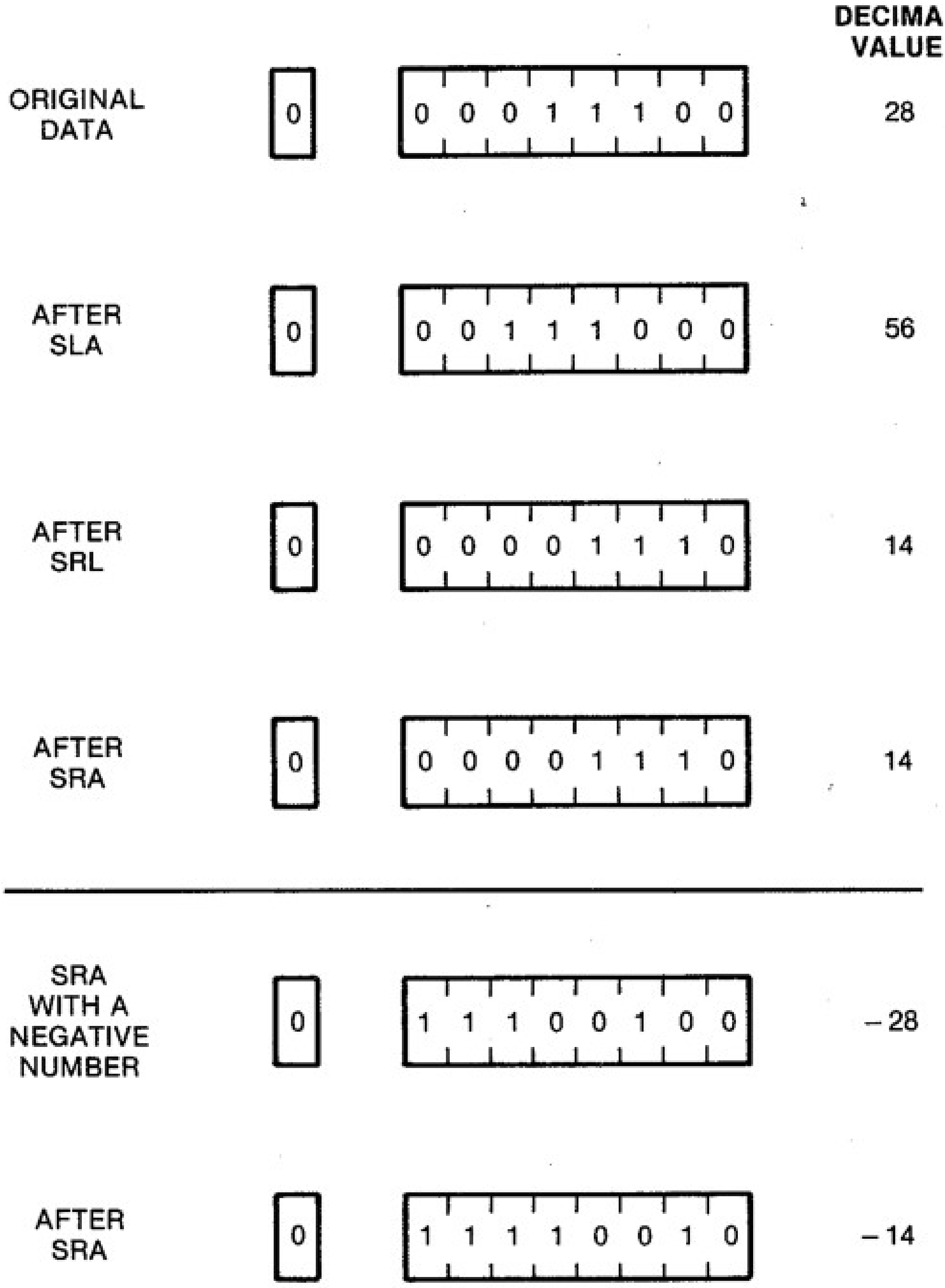


Fig. 12–5. Typical results of the SHIFT operations.

8-Bit Unsigned Integer Multiply

Listing 12–6 shows a routine which we have named UMULT. This program is written as a subroutine so that it can be called from other parts of a machine language program. UMULT takes the 8-bit *unsigned* value in register C and multiplies it by the 8-bit value in register B. Since an 8-bit by 8-bit multiply can yield a 16-bit product, we will store the result in the HL register pair. We can summarize the operation of this routine as follows:

Listing 12–6

```
; UNSIGNED 8 X 8 MULTIPLY

;MULTIPLIER IN C
;MULTIPLICAND IN B
;PODUCT IN HL

UMULT   LD E,8          ;E used as a counter
        LD HL,ØØØØ      ;Clear product register
LOOP    LD A,C          ;Get multiplier
        RRC A           ;Rotate right
        LD C,A          ;and re-save
        LD A,H
        JR NC,SHIFT     ;If not 1, then skip add
        ADD B           ;Add multiplicand
SHIFT   RRA             ;C → H(7) H(Ø) → C
        LD H,A          ;Save new H
        RRL A           ;H(Ø) → L(7)
        DEC E
        JR NZ,LOOP      ;Repeat for 8 bits
```

Listing 12–7

```
;SIGNED 8 X 8 MULTIPLY

SMULT   LD D,Ø           ;Clear product sign flag
        LD A,B           ;Get multiplicand
        ORA A            ;Set flags
        JR P,SM1         ;If positive, continue
        NEG              ;Form 2's complement
        LD B,A           ;and re-save
        INC D            ;Keep track of sign
SM1     LD A,C           ;Get multiplier
        ORA A            ;Do the same for
        JR P,SM2         ;multiplier
        NEG
        LD C,A
        INC D
SM2     CALL UMULT       ;Do multiply
        BIT Ø,D          ;Check LSB of D
        JR Z,DONE        ;Exit if product is positive
        LD A,H           ;Make 2's compliment
        CPL              ;of 16-bit product
        LD H,A
        LD A,L
        CPL
        LD L,A
        INC HL
DONE    RET              ;exit to caller
```

1. Initialize register E to a value of eight. This register is used as a counter to make one pass through the loop for each bit in the multiplier.

2. Clear out the product register (i.e., set HL = 0000) so that we can keep a running total of the partial products.
3. Start each pass of the loop by rotating out the next least significant bit of the multiplier. Since we want to get a new bit for each pass, we must re-save the rotated value back into the C register. We could also use a shift instead of rotate, but the latter preserves the contents of the multiplier.
4. Load A with the high-order byte on the running total.
5. Add the multiplicand to A *only* if the last bit rotated out from the multiplier was a 1.
6. Rotate right the entire 16-bit total.
7. Repeat the loop until done.

To give each partial product its proper weight, this program uses a rotate right on the running total instead of a rotate left on each partial product. You should be able to see that they are equivalent. Thus, on the first pass through the loop we get the least significant partial product. This is added to the most significant byte of the total but then is shifted one bit lower. After eight passes through the loop, and eight rotate rights, this product takes on its correct importance in the least significant byte.

8-Bit Signed Integer Multiply

To extend our program to signed numbers, we merely have to test the sign of each factor before multiplying. First, we determine what the sign of the product will be — positive if the signs are the same; negative if they are different. Then we convert any negative numbers to their positive value. At this point we can CALL the unsigned multiply routine UMULT to perform the actual multiplication. When this routine returns, the product will be in HL. If the result is supposed to be positive, then we are all done. Otherwise, we must take the twos complement of the product to convert it to the proper format.

Listing 12–7 shows the listing for the SMULT routine. Here the D register is used to hold the sign of the product. After performing the multiplication, the least significant bit of D is tested. If neither factor was negative, then D = 0, and, therefore, bit 0 of D is 0. Likewise, if both factors were negative, D would equal 2 and again bit 0 of D would be 0. If only one factor was negative, D would be 1 with bit 0 obviously set. In this case, we form the twos complement of the 16-bit result.

8-Bit Unsigned Integer Division Routine

Writing a division routine requires a few more considerations. First of all, there is the problem of attempting to divide by zero. This condition must be trapped out before getting to the actual division routine. Then, there is also a question as to the precision necessary for the quotient. This may require that one or both of the operands be scaled to put the quotient into a given range. Finally, there is a choice of how to perform the actual subtraction process. At each step in the calculation, we must detect whether the divisor is larger than the current dividend. If it is, then a one is added to the left of the quotient and the divisor is subtracted from the dividend. Otherwise a zero is added to the quotient and no subtraction is performed. Instead of checking the two numbers first, it is often simpler to perform the subtraction first and then check for an underflow. If this happens, then the current quotient position is made a zero and the divisor is added back in. This is known as a *restoring division* because the partial dividend is restored to its original value if the underflow occurs. Listing 12–8 shows a typical restoring division routine for the Z80.

Listing 12–8

```
;UNSIGNED 16 DIVIDED BY 8

;DIVIDEND (16 BITS) IN HL
;DIVISOR (8 BITS) IN D
;QUOTIENT (16 BITS) IN IX
;REMAINDER (8 BITS) IN H

DIV     LD A,L          ;Move dividend
        LD L,H
        LD E,Ø          ;Set for 16 bit SBC
        LD H,Ø          ;Clear remainder
        LD B,16         ;Loop 16 times
        LD IX,Ø         ;Clear quotient
LOOP    ADD HL,HL       ;Trick to shift left
        RLA A           ;Get next bit
        JP NC,SHIFT     ;Skip if zero
        INC L
SHIFT   ADD IX,IX       ;Shift quotient left
        INC IX          ;Set bit=1
        ORA A           ;Clear carry for SBC
        SBC HL,DE       ;Do subtract
        JP NC,NEXT      ;No underflow, continue
        ADD HL,DE       ;Otherwise restore
        DEC IX          ;Set bit=Ø
NEXT    DJNZ LOOP       ;Do 16 times
        RET             ;Exit when done
```

13

Special Instructions and I/O

Two of the most powerful features of the Z80 are its duplicate register sets and the block move/search instructions. This chapter will discuss how to use these advanced functions. We will also describe the INput and OUTput instructions which are used to transfer data through the I/O ports of the Z80. Then we'll finish the discussion of the Z80 instruction set by examining the general-purpose arithmetic and miscellaneous CPU control groups.

EXCHANGE GROUP : EX and EXX

As described in Chapter 4, the Z80 has two complete sets of general-purpose registers and a duplicate set of accumulator and flag registers. These *primed* registers can only be accessed by exchanging them with their *nonprimed* counterparts. Then data can be written to or read from these registers using the normal instructions. The A and F registers are always swapped together (AF ⟺ AF'). The general-purpose registers (BC, DE, and HL) are also exchanged as a single group. See Table 13–1 and Table 13–2 for details of the Z80 Exchange Group.

The Z80 also allows a limited number of exchanges to be performed between the special-purpose registers. The single byte op code EBh, for example, swaps the contents of the HL register with that in the DE register pair. Note that this is equivalent to the following code:

```
LD          TEMP,HL
LD          HL,DE
LD          DE,TEMP
```

where TEMP is a 16-bit temporary storage register or memory location. Obviously the EXchange instruction saves memory, executes much faster, and does not require the use of a temporary storage

Table 13–1. EXCHANGE Group (Courtesy MOSTEK, Corp.)

	Symbolic	Flags								Op-Code		No. of	No. of M	No. of T	
Mnemonic	Operation	S	Z		H		P/V	N	C	76 543 210	Hex	Bytes	Cycles	States	Comments
EX DE, HL	DE↔HL	•	•	X	•	X	•	•	•	11 101 011	EB	1	1	4	
EX AF, AF'	AF↔AF'	•	•	X	•	X	•	•	•	00 001 000	08	1	1	4	
EXX	BC↔BC' DE↔DE' HL↔HL'	•	•	X	•	X	•	•	•	11 011 001	D9	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔(SP+1) L ↔(SP)	•	•	X	•	X	•	•	•	11 100 011	E3	1	5	19	
EX (SP), IX	IX_H↔(SP+1) IX_L↔(SP)	•	•	X	•	X	•	•	•	11 011 101 11 100 011	DD E3	2	6	23	
EX (SP), IY	IY_H↔(SP+1) IY_L↔(SP)	•	•	X	•	X	•	•	•	11 111 101 11 100 011	FD E3	2	6	23	

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 13–2. Op Codes for EXCHANGE Group (Courtesy MOSTEK, Corp.)

		IMPLIED ADDRESSING				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		D9			
	DE			EB		
REG. INDIR.	(SP)			E3	DD E3	FD E3

location. Of course, only certain registers can be exchanged, so the preceding routine is useful as a general approach to exchanging data.

BLOCK TRANSFER GROUP : LDI, LDIR, LDD, LDDR

Refer to Table 13–3 and Table 13–4 as we discuss the block transfer instructions. These four instructions can be used to copy a contiguous range of memory locations from one place to another. There are no restrictions on the source or destination addresses nor on the number of bytes transferred. In many cases, the source and destination addresses will overlap. Before using any of these instructions, the HL, DE, and BC registers must first be initialized. The HL pair is loaded

Table 13–3. BLOCK TRANSFER Group (Courtesy MOSTEK, Corp.)

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
LDI	(DE)←(HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	X	0	X	① ↕	0	•	11 101 101 10 100 000	ED A0	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE)←(HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 110 000	ED B0	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	X	0	X	① ↕	0	•	11 101 101 10 101 000	ED A8	2	4	16	
LDDR	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 111 000	ED B8	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A – (HL) HL ← HL+1 BC ← BC-1	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 100 001	ED A1	2	4	16	
CPIR	A – (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 110 001	ED B1	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 13–4. Op Codes for BLOCK TRANSFER Group (Courtesy MOSTEK, Corp.)

			SOURCE	
			REG. INDIR.	
			(HL)	
DESTINATION	REG. INDIR.	(DE)	ED A0	'LDI' – Load (DE)←(HL) Inc HL & DE, Dec BC
			ED A0	'LDIR,' – Load (DE)←(HL) Inc HL & DE, Dec BC, Repeat until BC = 0
			ED B0	'LDD' – Load (DE)←(HL) Dec HL & DE, Dec BC
			ED B8	'LDDR' – Load (DE)←(HL) Dec HL & DE, Dec BC, Repeat until BC = 0

Reg HL points to source
Reg DE points to destination
Reg BC is byte counter

with the starting address of the source range, DE is loaded with the beginning of the destination range, and the number of bytes to be transferred is placed in BC. This is easy to remember if you think of DE for DEstination and BC for Byte Counter.

The LDI, or LoaD and Increment instruction, loads the data stored at the memory location pointed to by HL into the memory location pointed to by DE. Then the HL and DE registers are incremented and the BC register is decremented. After executing the LDI instruction, the program can determine if the entire range has been transferred (i.e., BC = 0) by testing the P/V flag. If this flag is 1, then there are still more bytes to transfer. The program can execute some more steps and then loop back to the LDI instruction to continue the block transfer. If the program does not need to perform any steps in between successive byte transfers, then the LDIR (LoaD, Increment, and Repeat) instructions can be used. This single instruction, all by itself, will transfer the entire range of bytes specified. As you can see from the symbolic representation, this instruction performs the LDI operation and, then, automatically repeats until BC = 0.

The LDD (LoaD and Decrement) and LDDR (LoaD, Decrement, and Repeat) instructions are similar to LDI and LDIR except that the registers are decremented after each transfer. This allows the transfer to proceed from the highest memory location to the lowest. While most of the time it does not matter in which order the transfer is made, there are some cases where it can be very important.

To simply transfer a block of memory from one range of addresses to another, nonoverlapping range, either LDI or LDD can be used. With LDI, the lowest memory address of the source range would go into HL, and the lowest address of the destination is put into DE. BC, of course, gets the total number of bytes in the block being transferred.

With LDD, the *highest* address of both ranges would be used; BC would stay the same. After executing either of these block moves, the results would be identical: a copy of the source range would be transferred into the destination range. See Fig. 13–1.

When the source and destination ranges overlap, however, things change considerably. For example, suppose we want to move a range of bytes up or down by one address. Except for one byte, every address in the destination range is also in the source range. Therefore, when we read data from a source location, it is important to make sure that the original data is still there. We would not want to retrieve data from this location that had already been transferred previously from

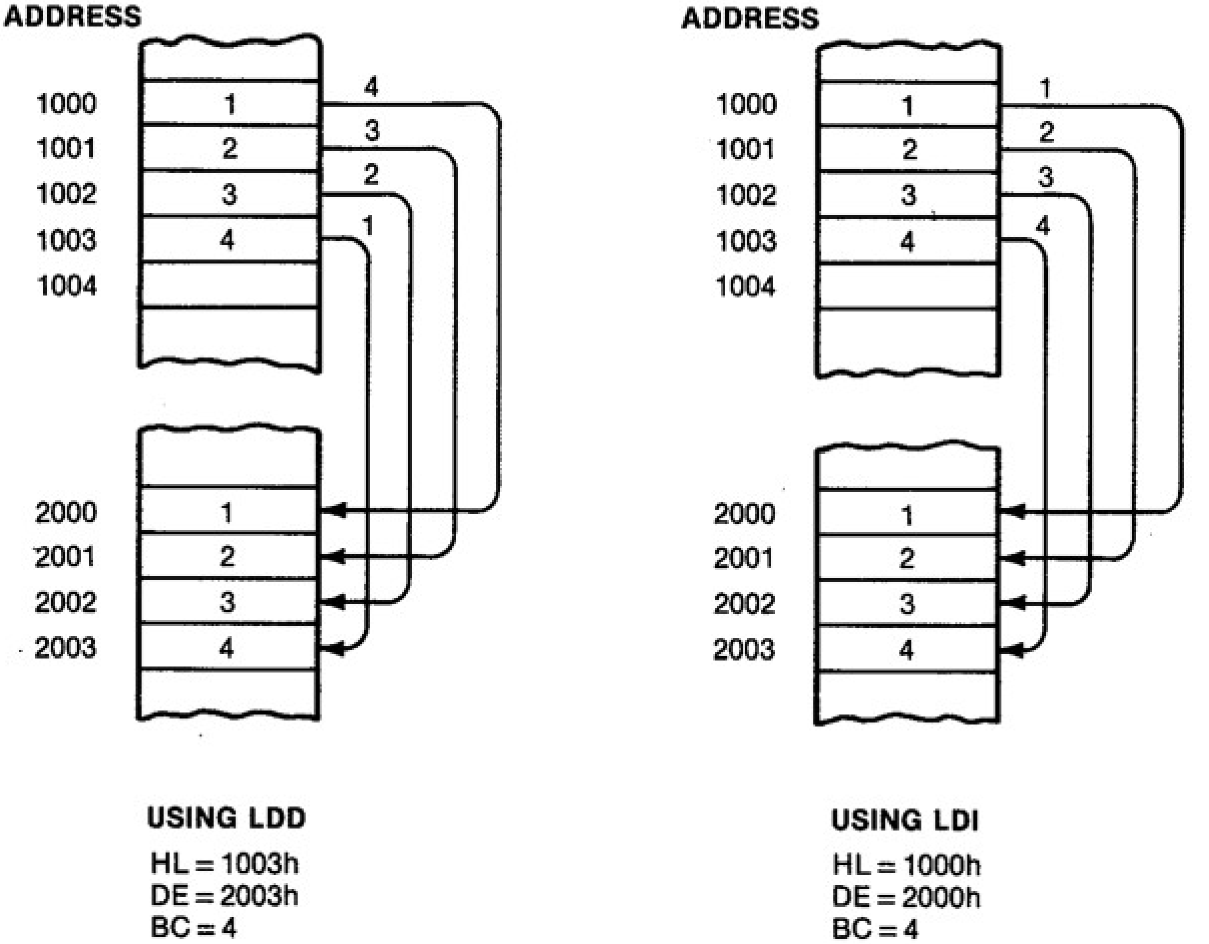


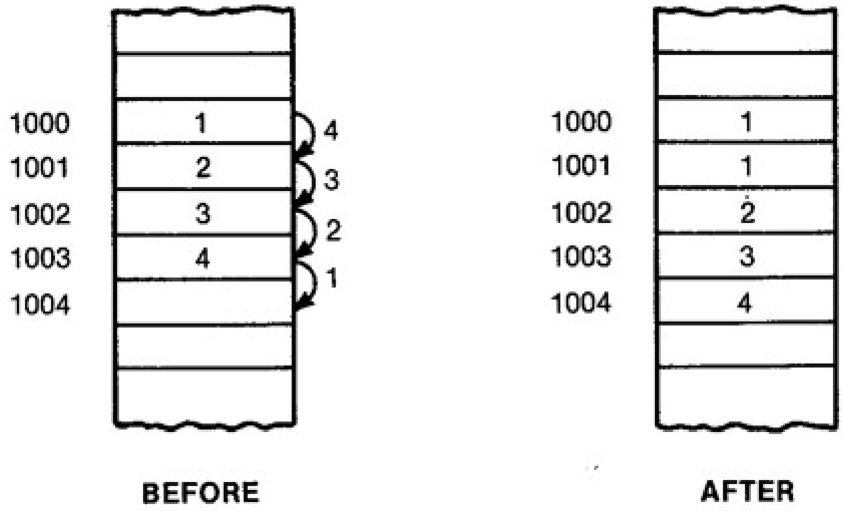
Fig. 13–1. Transferring a block of memory with nonoverlapping ranges.

another source location. Under these circumstances, the entire destination range would end up being filled with just one value — that of the initial source byte. Of course, this can also be a useful operation.

Fig. 13–2 shows the outcome of moving a block of memory up one byte using the LDDR instruction. The arrows indicate the memory transfers; they are numbered to show the order in which they take place. Note that we must start at the high end of the range and then work our way down for this operation to work properly. Had we started at the bottom of the range and used the LDIR instruction, we would have the results shown in Fig. 13–3.

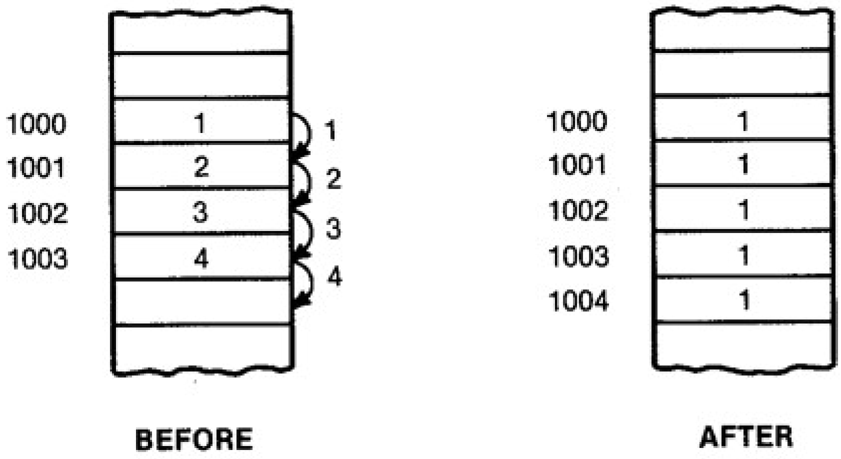
BLOCK SEARCH GROUP: CPI, CPIR, CPD, CPDR

Another set of instructions which acts on an entire range of locations is the Block Search Group. (See Table 13–5 and Table 13–6). These can be used to look through a range of bytes for the occurrence of a particular hex value. Like the block transfer group, the search can be



USING LDD
HL = 1003h
DE = 1004h
BC = 4

Fig. 13–2. Moving a block of memory up one byte using LDD.



USING LDI
HL = 1000h
DE = 1001h
BC = 4

Fig. 13–3. Moving a block of memory up one byte using LDI.

done manually or automatically and from either direction. To use these instructions, we first load HL with the starting address of the search and BC with the number of bytes to search. The Accumulator is loaded with the value that we are looking for.

Upon execution of the CPI (ComPare and Increment) instruction, the contents of the memory location pointed to by HL will be compared with the value in the accumulator. That is, the memory value is sub-

Table 13–5. BLOCK SEARCH Group (Courtesy MOSTEK, Corp.)

Mnemonic	Symbolic Operation	Flags S	Flags Z		Flags H		Flags P/V	Flags N	Flags C	Op-Code 76 543 210	Op-Code Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
CPD	A – (HL) HL ← HL-1 BC ← BC-1	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 101 001	ED A9	2	4	16	
CPDR	A – (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 111 001	ED B9	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 13–6. Op Codes for BLOCK SEARCH Group (Courtesy MOSTEK, Corp.)

SEARCH LOCATION

REG. INDIR. (HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR', Inc HL, Dec BC repeat until BC = 0 or find match
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC Repeat until BC = 0 or find match

HL points to location in memory
to be compared with accumulator
contents
BC is byte counter

tracted from the accumulator, and the flag register is set according to the results of this operation. The Z flag is used as always to specify whether the two bytes were equal. If they were, then the Z flag is set. Following the compare, the HL register is incremented and BC is decremented. The P/V flag serves a special purpose after a CPI instruction. It is set to 0 if the remaining value in the BC register is zero. Otherwise, P/V = 1.

The CPIR (ComPare, Increment, and Repeat) instruction automatically repeats until either a match is found or the byte count reaches zero. The flag register is affected in the same way as the CPI instruction. CPD (ComPare and Decrement) and CPDR (ComPare, Decrement, and Repeat) are the descending versions of the preceding group. All of the block move and search instructions are extremely useful when dealing with character strings. They allow us to move entire strings around or to search them for certain characters.

INPUT AND OUTPUT GROUP: IN, OUT

The Z80 has a fairly extensive set of instructions to communicate with external devices through the use of *ports.* As discussed in Chapter 5, an I/O port is just another way to get data into or out of the CPU. To access an I/O port, we use the IN and OUT instructions and supply a *port address.* Most of the Z80 I/O instructions use the C register to hold the 8-bit port address. During the I/O operation, the contents of this register are placed on the lower 8-bits of the address bus. The contents of the B register are placed on the upper 8-bits, so, in effect, we have register indirect addressing using the 16-bit BC pair.

This is how we can access over 65,000 different ports, if needed. Most Z80 computer systems find 256 I/O ports more than sufficient so the Z80 literature usually refers to only an 8-bit port address. In the T/S 2068, however, we have already seen that the joystick port uses a 9-bit address, and the entire 15 address bits are used for reading the keyboard. Let's take a closer look at the INput and OUTput instructions, as shown in Table 13–7 and Table 13–8.

The first instructions are the IN A,(n) and the corresponding OUT (n),A. These are the only two I/O instructions that do not use register indirect addressing. Instead, they use immediate addressing. That is, the byte following the op code is used to specify the port. This is used to form the lower half of the address bus with the accumulator going to the upper eight bits. This form of addressing can only be used to do I/O with the A register. Let's see how these instructions might be used to access one of the ports in the T/S 2068.

If you recall, the joystick inputs on the T/S 2068 are connected to the I/O port in the PSG. Therefore, to access these inputs, we must first address register 14 of the PSG. This is done by putting out 14 on the PSG address port (F5h). We then read the joystick signals on the PSG data port (F6h). But, to select which of the two joysticks we want, we

Table 13–7. INPUT and OUTPUT Group (Courtesy MOSTEK, Corp.)

Mnemonic	Symbolic Operation	Flags								Op-Code		No.of Bytes	No.of M Cycles	No.of T States	Comments
		S	Z		H		P/V	N	C	76 543 210	Hex				
IN A, (n)	A ← (n)	•	•	X	•	X	•	•	•	11 011 011 ← n →	DB	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	↕	↕	X	↕	X	P	0	•	11 101 101 01 r 000	ED	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INI	(HL) ← (C) B ← B - 1 HL ← HL + 1	X	① ↕	X	X	X	X	1	X	11 101 101 10 100 010	ED A2	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INIR	(HL) ← (C) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	X	X	1	X	11 101 101 10 110 010	ED B2	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
IND	(HL) ← (C) B ← B - 1 HL ← HL - 1	X	① ↕	X	X	X	X	1	X	11 101 101 10 101 010	ED AA	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INDR	(HL) ← (C) B ← B - 1 HL ← HL - 1 Repeat until B = 0	X	1	X	X	X	X	1	X	11 101 101 10 111 010	ED BA	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUT (n), A	(n) ← A	•	•	X	•	X	•	•	•	11 010 011	D3	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	X	•	X	•	•	•	11 101 101 01 r 001	ED	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTI	B ← B - 1 (C) ← (HL) HL ← HL + 1	X	① ↕	X	X	X	X	1	X	11 101 101 10 100 011	ED A3	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTIR	B ← B - 1 (C) ← (HL) HL ← HL + 1 Repeat until B = 0	X	1	X	X	X	X	1	X	11 101 101 10 110 011	ED B3	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTD	(C) ← (HL) B ← B - 1 HL ← HL - 1	X	① ↕	X	X	X	X	1	X	11 101 101 10 101 011	ED AB	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTDR	(C) ← (HL) B ← B - 1 HL ← HL - 1 Repeat until B = 0	X	1	X	X	X	X	1	X	11 101 101 10 111 011	ED BB	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$

Notes: ① If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

also have to set the ninth address bit (A8) accordingly. The following routine accomplishes this task with immediate mode I/O instructions:

```
LD A, 14           ;Select I/O register
OUT (F5h),A        ;on PSG
LD A, PLAYER       ;Set to either Ø or 1
IN A, (F6h)        ;Read joystick
```

Table 13–8. Op Codes for INPUT and OUTPUT Group

				PORT ADDRESS		
				IMMED.	REG. INDIR.	
				n	(C)	
INPUT DESTINATION	INPUT 'IN'	REG ADDRESSING	A	DB	ED 78	
			B		ED 40	
			C		ED 48	
			D		ED 50	
			E		ED 58	
			H		ED 60	
			L		ED 68	
	'INI' – INPUT & Inc HL, Dec B	REG. INDIR	(HL)		ED A2	BLOCK INPUT COMMANDS
	'INIR'– INP, Inc HL, Dec B, REPEAT IF B≠0				ED B2	
	'IND'– INPUT & Dec HL, Dec B				ED AA	
	'INDR'– INPUT, Dec HL, Dec B, REPEAT IF B≠0				ED BA	

			SOURCE								
			REGISTER							REG. IND.	
			A	B	C	D	E	H	L	(HL)	
'OUT'	IMMED.	n	D3 n								
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69		
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3	BLOCK OUTPUT COMMANDS
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3	
'OUTD' – OUTPUT Dec HL & B	REG. IND.	(C)								ED AB	
'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0	REG. IND.	(C)								ED BB	

PORT DESTINATION ADDRESS

BLOCK I/O: INI, INIR, IND, INDR, OUTI, OTIR, OUTD, OTDR

The Z80 is also capable of transferring a block of data through an I/O port. In this case, the HL register is loaded with the starting address of a memory range; the B register is used for a byte count. Since only the 8-bit B register is used, block I/O operations are limited to 256 bytes at one time. With the INI, IND, OUTI, and OUTD instructions, the Z flag is used to indicate when the B register has been decremented to zero (Z = 0 means that there are more bytes to transfer). Other than that, these instructions are used just like the block transfer commands.

GENERAL-PURPOSE ARITHMETIC GROUP: DAA, CPL, NEG, CCF, SCF

There are five general-purpose arithmetic instructions which operate on the accumulator or the carry flag. These are shown in Table 13–9 and Table 13–10. The DAA (Decimal Adjust Accumulator) instruction is useful when BCD arithmetic is performed. After performing addition or subtraction of BCD data, the result may not be in the correct form. Consider:

```
                 2                 6
          +-----------------+-----------------+
   26     |  Ø  Ø  1  Ø     |  Ø  1  1  Ø     |
          +-----------------+-----------------+
   +             3                 5
          +-----------------+-----------------+
   35     |  Ø  Ø  1  1     |  Ø  1  Ø  1     |
          +-----------------+-----------------+
 ______       ______________________________

                 5                ???
          +-----------------+-----------------+
          |  Ø  1  Ø  1     |  1  Ø  1  1     |  not valid BCD format
          +-----------------+-----------------+
```

Table 13–9. GENERAL-PURPOSE ARITHMETIC Group (Courtesy MOSTEK, Corp.)

	Symbolic	Flags								Op-Code		No. of	No. of M	No. of T	
Mnemonic	Operation	S	Z		H		P/V	N	C	76 543 210	Hex	Bytes	Cycles	States	Comments
DAA	Converts acc, content into packed BCD following add or subtract with packed BCD operands	↕	↕	X	↕	X	P	•	↕	00 100 111	27	1	1	4	Decimal adjust accumulator
CPL	$A \leftarrow \overline{A}$	•	•	X	1	X	•	1	•	00 101 111	2F	1	1	4	Complement accumulator (One's complement)
NEG	$A \leftarrow \overline{A}+1$	↕	↕	X	↕	X	V	1	↕	11 101 101 01 000 100	ED 44	2	2	8	Negate acc, (two's complement)
CCF	$CY \leftarrow \overline{CY}$	•	•	X	X	X	•	0	↕	00 111 111	3F	1	1	4	Complement carry flag
SCF	$CY \leftarrow 1$	•	•	X	0	X	•	0	1	00 110 111	37	1	1	4	Set carry flag

Table 13–10. Op Codes for GENERAL-PURPOSE ARITHMETIC Group (Courtesy MOSTEK, Corp.)

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

After executing the DAA instruction, however, we would have the correct result:

	6	1
61	Ø 1 1 Ø	Ø Ø Ø 1

The CPL (ComPLiment accumulator) instruction causes every bit in the accumulator to be changed. Thus:

Accum.	1	0	0	1	1	1	0	1

becomes after CPL

Accum.	0	1	1	0	0	0	1	0

The NEG (NEGate accumulator) instruction replaces the value in the accumulator with its twos complement notation. Therefore:

Accum.	1	0	0	1	1	1	0	1

becomes after NEG

Accum.	0	1	1	0	0	0	1	1

The last two instructions in this group operate only on the Carry flag. CCF (Complement Carry Flag) inverts the value of the flag while SCF (Set Carry Flag) forces the flag to 1. Although there is no direct instruction to reset the carry flag, this is easily accomplished through other single byte instructions such as AND A,A.

MISCELLANEOUS CPU CONTROL: NOP, HALT, DI, EI, IM0, IM1, IM2

Table 13–11 and Table 13–12 describe the instructions that control the operation of the Z80 and set the interrupt response mode. The first instruction is the NOP (No OPeration) which does just that — nothing. There are many reasons for having such an instruction. Probably the most common use for the NOP is to replace unnecessary or revised instructions in a previously written program. It is also useful for writing time-sensitive routines, since it will add a slight delay (one machine cycle) to the execution of a program.

The HALT instruction causes the CPU to stop executing its program until an interrupt is received. This is used to synchronize a program to external hardware. The T/S 1000, for example, uses such a scheme to let the Z80 perform the task of generating the video display.

Table 13–11. MISCELLANEOUS CUP CONTROL Group (Courtesy MOSTEK, Corp.)

	Symbolic	Flags								Op-Code		No. of	No. of M	No. of T	
Mnemonic	Operation	S	Z		H		P/V	N	C	76 543 210	Hex	Bytes	Cycles	States	Comments
NOP	No operation	•	•	X	•	X	•	•	•	00 000 000	00	1	1	4	
HALT	CPU halted	•	•	X	•	X	•	•	•	01 110 110	76	1	1	4	
DI*	IFF - 0	•	•	X	•	X	•	•	•	11 110 011	F3	1	1	4	
EI*	IFF - 1	•	•	X	•	X	•	•	•	11 111 011	FB	1	1	4	
IM 0	Set interrupt	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	8	
	mode 0									01 000 110	46				
IM 1	Set interrupt	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	8	
	mode 1									01 010 110	56				
IM 2	Set interrupt	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	8	
	mode 2									01 011 110	5E				

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

*Interrupts are not sampled at the end of EI or DI

Table 13–12. Op Codes for MISCELLANEOUS CUP CONTROL Group (Courtesy MOSTEK, Corp.)

'NOP'	00	
'HALT'	76	
DISABLE INT '(DI)'	F3	
ENABLE INT '(EI)'	FB	
SET INT MODE 0 'IM0'	ED 46	8080A MODE
SET INT MODE 1 'IM1'	ED 56	CALL TO LOCATION 0038_H
SET INT MODE 2 'IM2'	ED 5E	INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

The two instructions — DI (Disable Interrupt) and EI (Enable Interrupt) — control the software maskable interrupt facilities of the Z80. After executing the EI instruction, the internal IFF (Interrupt Flip Flop) flag is set, allowing an interrupt signal to be recognized. This flag will stay set until either the DI instruction is executed or an interrupt is received. An interrupt service routine should end with the EI instruction, if further interrupts are to be recognized. Since the Z80 does not check for an interrupt signal immediately following the EI instruction, if the next instruction is a RETI (Return from Interrupt), the routine will return properly without any danger of another interrupt routine starting before the first one ended. Of course, nested interrupts with varying priorities can be implemented if desired.

The IM0 (Interrupt Mode 0), IM1, and IM2 instructions are used to select how the Z80 will respond to interrupts. While a complete discussion of interrupts is beyond the scope of this book, we will give a brief description of the three modes available. The default condition is IM0 which allows the Z80 to handle interrupt requests like the 8080 CPU. Since the Z80 was designed to be an enhanced replacement for this early microprocessor, this mode is necessary for complete software compatibility. In the 8080 mode, the device generating the interrupt must place an instruction op code onto the data bus during the interrupt acknowledge time. This allows the device to "hardware program" the CPU to execute any instruction. This will usually be a restart instruction to a special interrupt handling routine somewhere in RAM.

In Mode 1, which the T/S 2068 uses, an interrupt causes a restart to location 38h. Thus, the interrupt handler would be loaded into that address. If the routine is of any great length, then a JuMP instruction to another area of memory may be used. Since the T/S 2068 uses interrupts to read the keyboard, there is a routine at location 38h to perform this task.

Mode 2 interrupts are used when the Z80 is attached to any of its peripheral chips. These devices support a comprehensive, vectored interrupt scheme which allows the interrupt to generate an indirect call to any location in memory. To accomplish this, the Z80 uses the I register as the upper 8 bits of an address and another 8-bit vector from the interrupting peripheral. This address points to an entry in a table of interrupt service routine addresses. Thus, the CPU reads the contents of this address, plus the next one, to find out the actual location of the service routine. The Z80 then performs a CALL to that address.

The Z80 also supports another interrupt structure called the NMI (Nonmaskable Interrupt). A different signal is used to generate an NMI, which *always* causes a restart to location 66h. There is no way to disable these interrupts.

With the completion of this chapter, we have described the entire Z80 instruction set. In the next section, we will show how machine language can be used from BASIC on the T/S 2068.

SECTION B
MACHINE LANGUAGE ON THE T/S 2068

14
Using Machine Language From BASIC

Now that we have covered the Z80 instruction set, we can begin to get serious about using machine language programs. Aside from writing the program itself on paper, we now have to consider how it will interact with the rest of the T/S 2068 hardware and software. First, we have to determine where in memory the program will reside. Then we have to physically place the program there. At the same time, we must ensure that the program will not interfere with the normal operation of the computer and vice versa. Consideration must be given to all of the computer's resources and how they are used. Memory locations, register usage, bank switching status, etc., must all be examined. Finally, don't overlook such hidden traps as the presence of external devices or the computer's interrupt generation (60 times per second to read the keyboard).

CREATING A MACHINE LANGUAGE PROGRAM

Obviously the first thing we have to do is to create the machine language program that we wish to execute. For example, suppose we want to fill the screen with a constant dot pattern. We know from Chapter 8 that this involves filling the memory locations in D_FILE_1 with a constant byte. This could be written in BASIC as follows:

```
10 FOR i = 16384 TO 22528 : POKE i , 51 : NEXT i
```

When you run this program, the screen will slowly fill up with the binary pattern represented by the data in the POKE statement. We chose the value 51 because in binary it becomes 00110011. Thus the screen will take on a pattern of vertical lines 2 pixels wide with 2 pixels in-between. You might also want to try other values such as 85, 17, or 7.

With a little playing around, you should be able to verify the structure shown in Fig. 8–5. This should be especially evident by the order in which the screen gets filled.

No matter what value you POKE in, the preceding program takes about 48 seconds to completely fill the screen. This is a classic candidate for replacement by a machine language routine. Such a program might look like Listing 14–1. This is the block move routine that is set up to replicate the first byte into an entire memory range. This first byte is defined by the second line of the routine which shows this value in binary form.

CHOOSING THE RIGHT LOCATION

The next thing we must consider is where to put the machine code. While BASIC will let us POKE the program almost anywhere in RAM, we must not do so indiscriminately. As we saw in Fig. 5–2, much of the computer's memory is dedicated to a particular use. We certainly would not want to POKE the program into the middle of the machine stack or on top of our BASIC program. If we examine this figure carefully, we will find that there are five distinct sections of memory. The stationary RAM, which is used by the system software, runs from location 16384 (D_FILE_1) to location 26709 (PROG − 1). This includes the display and attribute file, machine stack, bank switching code, etc. These data structures are always present (with minor changes when D_FILE_2 is used), and therefore represent a fixed overhead on the available RAM. The area of memory between PROG and E_LINE −1 represents the storage of our BASIC program and its variables. This, of course, will constantly be changing, and it can grow until all of the available RAM is used up. Directly above the variable storage, we find another series of system data structures. These include the edit buffer and the temporary work space for the BASIC

Listing 14–1

```
LD HL, 16384           ;Start of D_FILE_1
LD (HL),ØØ11ØØ11       ;Pattern byte to replicate
LD DE,16385            ;First byte to transfer
LD BC,6143             ;Length of D  FILE_1
LDIR                   ;Go to it!
RET                    ;Return to BASIC
```

interpreter. This section of memory, from E_LINE to STKEND, also moves around as the computer is used. With the exception of a small area at the very top of memory, the remaining RAM from STKEND to RAMTOP is unused. After RAMTOP, we have the data for the user-defined graphics characters. This continues up to the very last byte of RAM.

In looking for a place to put a machine language program, we can see that almost everything below STKEND is being used. There are a few tricky places to put machine code, such as in the area reserved for the printer buffer (if you are not using a printer, then these 256 bytes can be used for anything you desire). In general, however, we will want to place machine language programs above STKEND, into the unused area.

Since this free RAM gets eaten up from the bottom end as memory usage grows, it makes sense to place the routines as high as possible. This means that they should go just below the User Defined Graphics area. This is, in fact, where most machine language routines will be placed; we will show how to protect this area from BASIC.

In Chapter 12, we created some programs to run at location 28672 (7000h). This location was chosen because it was a nice round hex address and we could be sure that it was in the unused area. We knew that it was above STKEND because we only had a short BASIC program and a couple of variables being stored in memory. Since STKEND is initially set at location 26726, we knew that the program/variable usage would have to reach almost 2000 bytes before memory locations 28672 and above were needed. Thus, we were safe in putting our program there.

There is still one more problem to deal with, however. No matter where in free memory that we place the routine, there is always the chance that the BASIC program or variables will expand to that point. Since BASIC has no idea that we have something meaningful at these addresses, it will blindly overwrite these locations with whatever data it needs to store. So the machine language program could get destroyed if we are not careful.

Fortunately, there is a simple way to inform the BASIC interpreter that we do not want it to use memory locations above a certain address. This is the function of the system variable RAMTOP. It represents to BASIC the last byte of memory available to the interpreter. Thus, if we can move RAMTOP down in memory, we will create a free area of RAM between RAMTOP and UDG. We can then place machine language routines in this area where they will be safe from BASIC. While it is possible to change RAMTOP by POKEing in a new value, it is easier to use the CLEAR statement in BASIC. This statement automatically resets RAMTOP to the address specified by the command. Thus:

```
10 CLEAR 65000
```

will set RAMTOP to address 65000. Since RAMTOP is normally 65367 on the T/S 2068, this would give us 367 bytes of free memory in which to place the routines. With RAMTOP set at 65000, the first byte available for machine code is at location 65001.

ASSEMBLING THE PROGRAM

Now that we know where the program is going, we will return to our example and hand-assemble it. We start by converting the starting address to hex (65001 = FDE8h). Then we look up the op code for each instruction and convert any binary or decimal operands to hex. This gives us the complete hex representation, or *object code* for the program. Finally, we convert each byte of hex code to its decimal equivalent (jot them down on the right side), and we're ready to POKE the program into memory. After assembly, the program now looks like Listing 14–2.

Listing 14–3 shows how we take the decimal values, placed in a DATA statement, and POKE them into the locations starting above the new RAMTOP setting. When this program is run, the screen fills up in about one-half of a second. That's 100 times faster than the BASIC version! In fact, most of this half-second is still being used up by the interpreter executing the first five lines. Type:

```
RUN 100
```

Listing 14–2

```
FDE8- 21 ØØ 4Ø       LD HL, 16384          33,Ø,64
FDEB- 36 33          LD (HL),ØØ11ØØ11      54,51
FDED- 11 Ø1 4Ø       LD DE,16385           17,1,64
FDFØ- Ø1 FF 17       LD BC,6143            1,255,23
FDF3- ED BØ          LDIR                  237,176
FDF5- C9             RET                   2Ø1
```

Listing 14–3

```
 1Ø   CLEAR 65ØØØ
 2Ø   FOR a=65ØØ1 TO 65Ø14
 3Ø   READ d: POKE a,d
 4Ø   NEXT a
 5Ø   DATA 33,Ø,64,54,51,17,1,64,1,255,23,237,176,2Ø1
1ØØ   RANDOMIZE USR 65ØØ1
```

to see how fast the machine language program really is.

Once we have run the program, the machine language routine will be safely stored in memory. Typing NEW will wipe out any existing BASIC program, but the machine language routine will remain. Try it:

```
NEW
RANDOMIZE USR 65ØØ1
```

CASSETTE STORAGE OF MACHINE LANGUAGE ROUTINES

Once we have a working machine language program, we may want to save it on cassette. Before we typed in NEW, the BASIC program that POKE'd it in was still in memory. This program contains the machine language routine as a series of numbers in the DATA statement. Obviously, it can be saved on cassette, reloaded back in, and then RUN to execute the routine. When the machine code is less than 50 – 100 bytes, this method works fine.

With longer routines, we begin to suffer a few problems. First, it should be noted that the DATA statement representation of the code requires several times as many bytes as the actual code. This is because each hex byte will require eight to ten bytes when placed in the DATA statement (1–3 bytes for the decimal digits, 6 bytes for the "number" code and its 5-byte value, and another byte for the comma between each data element).

Thus a moderately long machine language program will require a very long BASIC program to POKE it in. This means that the BASIC program will also take quite a long time to load in from cassette. The loop which does the POKEing can also take a considerable amount of time when the program is RUN. To make matters even worse, after the routine has been POKE'd in, there are actually two copies of the same program taking up memory space. The original data (which takes up the most room) is no longer needed so it is wasting space.

This can be solved by DELETEing the program lines which do the POKEing along with the associated DATA statements.

A much better approach to saving and loading machine language programs is to use the special forms of the SAVE and LOAD commands. If you type:

```
SAVE "screen" CODE 65ØØ1,14
```

then the machine language routine will be saved on tape (even though it's still hidden from BASIC). The SAVE . . . .CODE command allows you to "dump" any portion of the T/S 2068's memory, in binary form, to the cassette.

To retrieve the program, we would type:

```
LOAD "screen" 65ØØ1,14
```

This can be done in the command mode or from any point within a BASIC program. Since we usually want to use the machine code in conjunction with a BASIC program, it makes sense to have them both loaded in at the same time. This can be accomplished by having one of the first lines of the BASIC program do the LOAD . . . CODE operation. Under these circumstances, you would also want to have the BASIC program begin executing as soon as it is loaded. With our example, we could do this by writing a new BASIC program:

```
10 LOAD "screen" 65ØØ1,14
20 RANDOMIZE USR 65ØØ1
```

Then we would save it onto cassette with:

```
SAVE "demo" LINE 1Ø
```

and as soon a the report code appears, type:

```
SAVE "screen" 65ØØ1,14
```

This will place both parts of the program next to each other. If you now rewind the tape and type:

```
LOAD "demo"
```

the screen demo program will automatically start executing.

15
Using the Built-in ROM Routines

This chapter is for the adventurers out there who really like to dig into their computers. One of the most exciting things you can do with a computer is to poke around (or should that be PEEK around?) with its ROM program attempting to discover some useful subroutines. Many times, you will find some interesting POKE's or USR calls that allow you to do things not possible from BASIC. Or you might discover one of the interpreter's routines that can be accessed directly — even from BASIC — to perform a given function. This information can make your BASIC programs run much faster and your machine language programs easier to write.

PRECIOUS POKES

Probably the best place to go looking for POKES is in the Table of System Variables found in Appendix D of the *Timex Sinclair 2068 User Manual*. By now, some of these cryptic descriptions should make a little sense. Let's pick a few that have some really useful function. Memory location 23609 is called PIP because it holds a value that determines the sound that the T/S 2068 makes when a key is pressed. When the computer is first turned on, this location is initialized with a value of one. This causes the very short click that you normally hear when pressing a key. If you change this value by entering:

```
POKE 23609,128
```

you will find that the key "click" has now become a key "beep." Actually, it is the length of this beep that is stored in location 23609. Therefore, when the computer initializes this byte to 1, it really makes only one cycle of the tone which sounds like a click.

Another interesting location to POKE at is address 23562. This is called REPPER, and it controls the speed of the auto-repeat feature on the keyboard. The value at this location represents the delay — in 1/60ths of a second — between successive repeats when a key is held down. Initially it is 5 but you can change this to any number from 1 to 225. Try typing:

```
POKE 23562,1
```

and then hold down the space bar. One thing to watch out for: this function is somewhat tied to the previous one. If the PIP value is very large, then it takes a considerable amount of time for each beep to be produced. The computer will not accept another keystroke until it has finished sounding the previous one. Therefore, it is possible that the PIP value will override the REPPER rate setting. By the way, location 23561 holds a value called REPDEL which sets the time that a key must be held down before it begins to repeat. You cannot hurt the computer in any way by changing the values of these system variables. The worst that can happen is that the computer will hang up, and you will have to turn it off and back on again to regain control. So be adventurous!

USING USR

With most computers, the real hacker would next begin to explore various ROM routines. Finding out where they are, what they do, and what else they affect can be fun, but it also takes a lot of time. In the case of the T/S 2068, there is another point to consider. At the time this book is being written, there is a strong possibility that the ROMs in the computer will undergo some changes. Therefore, any routines that we might document here could well end up being obsolete in the near future. This is one of the negative aspects of using ROM routines in your programs. Fortunately, however, Timex has incorporated a special section in the ROM which makes everything much easier.

THE FUNCTION DISPATCHER

Built-in to the T/S 2068 Operating System ROM is a mechanism for handling all of the basic functions of the computer. During initialization, this ROM installs a special utility program into RAM at location 6200h (or FA50h if D_FILE_2 is being used). This utility, called the Function Dispatcher, forms a consistent interface between application or system software and the rudimentary operations needed to control the T/S 2068 hardware.

Table 15–1 lists all of the functions currently available to the Function Dispatcher.* Note that each function is identified by a service code number. To use the Function Dispatcher, we must first initialize the machine stack and then PUSH a two-byte value representing the service code. Any other registers used by the particular function are then set up. Finally, a call is made to the Dispatcher which then performs the desired task. It does this by CALLing certain other routines stored in ROM.

As an example, let's see how we can plot a point on the screen using the PLOTBC routine. This routine performs the equivalent of the BASIC statement PLOT C,B. There are also four variations of this command, depending upon the value stored in the system variable PFLAG (23697). Bit 0 of this variable is called XOR-CH; bit 2 is called INV-CH. Table 15–2 outlines the significance of these bits to the PLOTBC function. Normally this routine will cause a pixel to be set.

Listing 15–1 shows a machine language routine that uses the Function Dispatcher and PLOTBC to draw a vertical line on the screen. This involves plotting 174 separate points from within a loop. Since the B register holds the vertical coordinate for the PLOTBC operation, we use the DJNZ instruction to create the loop. We, therefore, start by initializing the B register to 175 and loading the C register with the desired X-coordinate (in this case 127). We can load both registers simultaneously using the LD BC instruction.

Next, we begin our loop by saving the BC register on the stack. This is because the contents of this register will get changed after calling the Function Dispatcher. We then need to PUSH two sets of 2-byte zeros to set up the stack for the Dispatcher. We PUSH the service code onto the stack and we're ready to CALL the Function Dispatcher. After plotting the point indicated by register B and C, the routine returns to our POP BC instruction. This recovers the coordinates of the point just plotted. The DJNZ instruction lowers the Y-coordinate by one and then jumps back to plot the next point. After 174 points have been

*As of this writing, it is known that some of these functions (such as CHNG_VID) do not work. These problems may be corrected in future ROM revisions.

Table 15–1. T/S 2068 Functions

Function	Service Code	Action
W_TAPE	Ø	Write a block to tape.
R_TAPE	1	Read a block from tape.
RD_BIT	2	Read a bit from tape.
R_EDGE	3	Read an edge from tape.
SLVM	4	General tape routine.
LOAD	5	Load.
MERGE	6	Merge.
SAVE	7	Save.
CHNG_VID	8	Change video mode.
W_BORD	9	Write border color.
	1Ø	Reserved.
	11	Reserved.
	12	Reserved.
	13	Reserved.
GET_STATUS	14	
GET_NUMBER	15	
BANK_ENABLE	16	
GOTO_BANK	17	
CALL_BANK	18	
XFER_BANK	19	
	2Ø	Reserved.
	21	Reserved.
	22	Reserved.
	23	Reserved.
	24	Reserved.
UPD_K	25	Scan keyboard.
PARP	26	Sound routine.
BEEP	27	BEEP command.
K_DUMP	28	COPY command.
SENDTV	29	Send char. to screen.
SETAT	3Ø	Set print position.
STTBYT	31	Fix attribute byte.
R_ATTS	32	Temp. Atts., Perm. Atts.
CLLHS	33	Clear lower half of screen.
CLS	34	Clear entire screen.

Table 15–1 — cont. T/S 2068 Functions

Function	Service Code	Action
DUMPPR	35	Printer buffer sent to print.
PRSCAN	36	Send scan to printer.
DESLUG	37	Remove slugs from line buffer.
K_NEW	38	NEW command.
INIT	39	Initialize.
INCH	4Ø	Input character.
SELECT	41	Select current stream.
INSERT	42	Insert bytes.
RESET	43	Reset calculator stack.
CLOSE	44	CLOSE command.
CLCHAN	45	Close channel.
OPEN	46	OPEN command.
OPCHAN	47	Open channel.
CAT	48	CAT command.
DELETE	49	DELETE command.
FORMAT	5Ø	FORMAT command.
MOVE	51	MOVE command.
FLASHA	52	Flash char. in A to screen.
FINDL	53	Find BASIC line.
SUBLIN	54	Find sub-line.
RECLEN	55	Record length.
DELREC	56	Delete record.
PUT_LN	57	Send line to output.
SYNTAX	58	Check syntax.
EXECUTE	59	Execute line.
FOR	6Ø	FOR command.
STOP	61	STOP command.
NEXT	62	NEXT command.
READ	63	READ command.
DATA	64	DATA command.
RESTBC	65	RESTORE bc.
RAND	66	RAND command.
CON'T	67	CON'T command.
JUMP	68	Jump to line.
FIX_U1	69	Fix 1 byte # from calc. stack.

Table 15–1 — cont. T/S 2068 Functions

Function	Service Code	Action
FIX_U	7Ø	Fix 2 byte # from calc. stack.
CLEAR	71	CLEAR command.
CLR_BC	72	CLEAR bc.
GO_SUB	73	GOSUB command.
CHKUSR	74	Free space left.
RETURN	75	RETURN command.
PAUSE	76	PAUSE command.
BREAK?	77	Break key pressed?
DEF	78	DEF command.
K_LPR	79	LPRINT command.
K_PRIN	8Ø	PRINT command.
P_SEQ	81	Print sequence.
INPUT	82	INPUT command.
I_SEQ	83	Input sequence.
NOTKB?	84	Test CURCHL=KB.
COLOR	85	Adjust attributes sysvars.
HIFLSH	86	Adjust attributes sysvars.
SCRMBL	87	Screen address calculator.
PLOT	88	PLOT command.
PLOTBC	89	Plot c, b.
GET_XY	9Ø	Get x and y.
CIRCLE	91	CIRCLE command.
DRAW	92	DRAW command.
DRAW_L	93	Draw line.
EXPRN	94	Expression Evaluator.
F_SCRN	95	Run time action for SCREEN $.
F_ATTR	96	Run time action for ATTR.
RND	97	RND action.
F_PI	98	PI action.
F_INKY	99	INKEY action.
FIND_N	1ØØ	Find variable.
PSHSTR	1Ø1	Push string.
PAEDCB	1Ø2	Push A, E, D, C, B.
LET	1Ø3	LET command.
POPSTR	1Ø4	Pop string.

Table 15–1 — cont. T/S 2068 Functions

Function	Service Code	Action
DIM	1Ø5	DIM command.
STKUSN	1Ø6	Stack unsigned number.
STK_A	1Ø7	Stack A.
STK_BC	1Ø8	Stack BC.
ININT	1Ø9	Read/Stack Integer.
FP2BC	11Ø	Get 16 bit #.
FP2A	111	Get 8 bit #.
OUTPUT	112	Output number on stack.
SUB	113	Subtract.
ADD	114	Add.
MULT	115	Multiple (int).
TIMES	116	Multiply (F.P.).
DIVIDE	117	Divide.
TRUNC	118	Truncate.
FLOAT	119	Force Int F.P.
INTDIV	12Ø	Integer Divide.
INT	121	INT.
EXP	122	EXP.
LN	123	LN.
ANGLE	124	Angle calculator.
COS	125	COS.
SIN	126	SIN.
TAN	127	TAN.
ATN	128	ATN.
ASN	129	ASN.
ACS	13Ø	ACS.
ROOT	131	SQR calculator.
TO_THE	132	Exponentiation.
RDCH	133	Read character.
SENDCH	134	Send character.
WRCH	135	Write character.
K_SCAN	136	Keyboard scan.
P_LFT	137	Print cursor left.
P_RT	138	Print cursor right.
P_NL	139	Print newline.

Table 15–1 — cont. T/S 2068 Functions

Function	Service Code	Action
PUTMES	14Ø	Print message.
K_CLS	141	CLS command.
SCRL	142	Scrolling routine.
F_PNT	143	Point action.

Table 15–2. PLOTBC Options

XOR-CH	INV-CH	Function
Ø	Ø	Set pixel.
Ø	1	Reset pixel.
1	Ø	Invert pixel.
1	1	Leave pixel as is.

plotted, this routine returns to BASIC. By using the DJNZ instruction, we do not plot a point at the zero Y-coordinate location.

Listing 15–2 shows the BASIC program to enter this routine into memory. After RUNning this BASIC program, type:

```
RANDOMIZE USR 65ØØ1
```

to RUN the routine. Note that it takes about one-half second to execute.

Listing 15–1

```
FDE8- Ø1 AF 7F          LD BC,7FAF    ;Start at (127,175)     1,127,175
FDEB- C5          LOOP  PUSH BC       ;Save coordinates       197
FDEC- 11 ØØ ØØ          LD DE,ØØØØ    ;Set up                 17,Ø,Ø
FDEF- D5                PUSH DE       ;Function               213
FDFØ- D5                PUSH DE       ;Dispatcher             213
FDF1- 11 27 ØØ          LD DE,89      ;Get Service Code       17,89,Ø
FDF4- D5                PUSH DE       ;and push it            213
FDF5- CD ØØ 62          CALL 62ØØh    ;Go to Func. Disp.      2Ø5,Ø,98
FDF8- C1                POP BC        ;Restore coordinates    193
FDF9- 10 EF             DJNZ,LOOP     ;Do another point       16,239
FDFB- C9                RET           ;Exit when done         2Ø1
```

Listing 15–2

```
1Ø   CLEAR 65ØØØ
2Ø   FOR a=65ØØ1 TO 65Ø2Ø
3Ø   READ d: POKE a,d
4Ø   NEXT a
5Ø   DATA1,127,175,197,17,Ø,Ø,2,13,213,17,89,Ø,213,2Ø5,Ø,98,193,16,239,2Ø1
```

If you want to compare this to the equivalent BASIC program, type NEW and then:

```
1Ø FOR i=175 TO 1 STEP -1: PLOT 127,i: NEXT i
```

This program takes 1.6 seconds to execute. In this case, the Function Dispatcher is adding some overhead to the machine language routine but it is still over three times faster. Of course, when you compare the machine language program of Listing 15–1 to the BASIC equivalent in terms of simplicity and readability, BASIC wins hands down.

There you have the great compromise. Machine language programs will outrun higher level programs such as those written in BASIC. But the machine language programs will take considerably longer to write (and even longer to debug!). Machine language programs are also less easily modified. Having a knowledge of both BASIC and machine language will allow you to combine the best features of each according to your needs. The following paragraphs describe some of the other functions provided by the Function Dispatcher.

W_Tape (Service Code Ø)

This routine writes a block of data from memory to the cassette tape. Upon entry, the IX register should point to the starting address of the block and the DE register should specify the number of bytes to be written. Before beginning the block transfer, the contents of the A register will be written out; it should be a 0 for data or a FFh to indicate that the following bytes are header information. The header block contains information about a file such as its name and length.

The data recorded on tape starts off with a leader signal. This is an 800-Hz tone which lasts for about 5 seconds on a header block or about 2 seconds for a data block. Following the leader signal is one cycle of 2040-Hz tone and then the data. Each data byte is sent out serially, MSB first. A zero bit is indicated by one cycle of 2040 Hz, and a one bit is signaled by one cycle of 1020-Hz tone. After all the data bits have been sent, a parity byte is added. This byte represents the Exclusive-OR of all the previous data bytes.

R_TAPE (Service Code 1)

This routine reads a block of data from the cassette. The number of bytes to be read is specified in the DE register. The IX register points to where the first byte will go in memory. The A register should also be set to a 0 or FFh to indicate what type of block is to be read. A running checksum is kept as the data is read in. This gets compared to the checksum byte recorded on the tape. This routine will return with the carry flag set if there were no errors. It returns with carry reset if the checksums did not match, if there were less than DE bytes read in, or if the block type was wrong.

BREAK? (Service Code 77)

This routine looks directly at the keyboard to determine if both the CAPS SHIFT and SPACE keys are being pressed (i.e., a BREAK). It returns with carry reset if this is true; otherwise, it returns with carry set.

SCRMBL (Service Code 87)

This is a screen address calculator. Since the pixel locations on the screen are stored in a scrambled form in memory, this routine is available to help unscramble it. Upon entry, the BC register should hold the PLOT-type coordinates. B should be holding Y and C should contain X. When this function returns, the HL register will be pointing at the byte in the display file where this pixel is stored. The A register will indicate the bit position within this byte.

APPENDICES

APPENDIX A
Powers of 2

2^N	N	2^{-N}
1	0	1
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25

APPENDIX B
Hex to Decimal Conversion

Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec
00	0	21	33	42	66	63	99
01	1	22	34	43	67	64	100
02	2	23	35	44	68	65	101
03	3	24	36	45	69	66	102
04	4	25	37	46	70	67	103
05	5	26	38	47	71	68	104
06	6	27	39	48	72	69	105
07	7	28	40	49	73	6A	106
08	8	29	41	4A	74	6B	107
09	9	2A	42	4B	75	6C	108
0A	10	2B	43	4C	76	6D	109
0B	11	2C	44	4D	77	6E	110
0C	12	2D	45	4E	78	6F	111
0D	13	2E	46	4F	79	70	112
0E	14	2F	47	50	80	71	113
0F	15	30	48	51	81	72	114
10	16	31	49	52	82	73	115
11	17	32	50	53	83	74	116
12	18	33	51	54	84	75	117
13	19	34	52	55	85	76	118
14	20	35	53	56	86	77	119
15	21	36	54	57	87	78	120
16	22	37	55	58	88	79	121
17	23	38	56	59	89	7A	122
18	24	39	57	5A	90	7B	123
19	25	3A	58	5B	91	7C	124
1A	26	3B	59	5C	92	7D	125
1B	27	3C	60	5D	93	7E	126
1C	28	3D	61	5E	94	7F	127
1D	29	3E	62	5F	95	80	128
1E	30	3F	63	60	96	81	129
1F	31	40	64	61	97	82	130
20	32	41	65	62	98	83	131

Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec
84	132	A3	163	C2	194	E1	225
85	133	A4	164	C3	195	E2	226
86	134	A5	165	C4	196	E3	227
87	135	A6	166	C5	197	E4	228
88	136	A7	167	C6	198	E5	229
89	137	A8	168	C7	199	E6	230
8A	138	A9	169	C8	200	E7	231
8B	139	AA	170	C9	201	E8	232
8C	140	AB	171	CA	202	E9	233
8D	141	AC	172	CB	203	EA	234
8E	142	AD	173	CC	204	EB	235
8F	143	AE	174	CD	205	EC	236
90	144	AF	175	CE	206	ED	237
91	145	B0	176	CF	207	EE	238
92	146	B1	177	D0	208	EF	239
93	147	B2	178	D1	209	F0	240
94	148	B3	179	D2	210	F1	241
95	149	B4	180	D3	211	F2	242
96	150	B5	181	D4	212	F3	243
97	151	B6	182	D5	213	F4	244
98	152	B7	183	D6	214	F5	245
99	153	B8	184	D7	215	F6	246
9A	154	B9	185	D8	216	F7	247
9B	155	BA	186	D9	217	F8	248
9C	156	BB	187	DA	218	F9	249
9D	157	BC	188	DB	219	FA	250
9E	158	BD	189	DC	220	FB	251
9F	159	BE	190	DD	221	FC	252
A0	160	BF	191	DE	222	FD	253
A1	161	C0	192	DF	223	FE	254
A2	162	C1	193	E0	224	FF	255

APPENDIX C
ASCII Character Set
(7-Bit Code)

	MSD	0	1	2	3	4	5	6	7
LSD		000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	″	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENG	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	′	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	′	<	L	\	l	/
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	●	>	N	∧	n	~
F	1111	SI	VS	/	?	O	—	o	DEL

Index

SAMS

Timex® Sinclair 2068 Intermediate/Advanced Guide

This second book in a series on the Timex Sinclair 2068 takes the reader beyond BASIC into the world of the computer's circuits, its microprocessor, and machine language. This book is written for the computer user with a basic understanding of programming who wants to learn what really makes his Timex tick. It covers

- Interfacing peripheral devices
- Machine language programs and subroutines
- Technical details of the Z80 microprocessor
- Information on using the Programmable Sound Generator
- Inner workings of the BASIC interpreter
- Memory mapping

The *Timex Sinclair 2068 Intermediate/Advanced Guide* teaches you the tricks of the programming trade—how to make everyday programs run faster, and how to make the 2068 do things you thought were impossible.

Howard W. Sams & Co., Inc.
A Publishing Subsidiary of ITT
4300 West 62nd Street, Indianapolis, Indiana 46268

$9.95/22226

ISBN: 0-672-22226-4