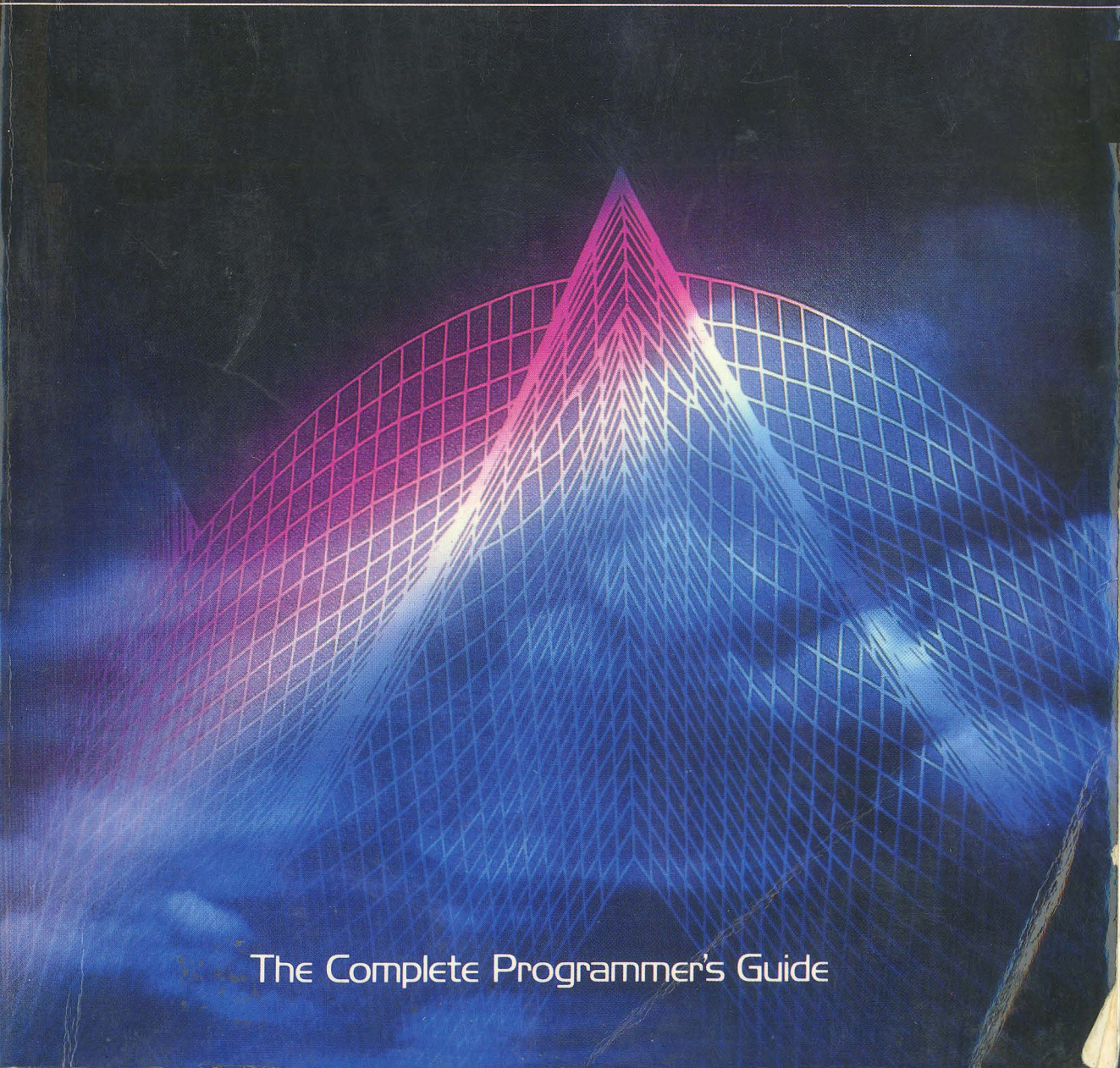
JAMES SUTTON

POWER PROGRAMMING THE COMMODORE 64

ASSEMBLY LANGUAGE, GRAPHICS AND SOUND



The Complete Programmer's Guide

JAMES SUTTON

POWER PROGRAMMING THE COMMODORE 64: ASSEMBLY LANGUAGE, GRAPHICS, AND SOUND

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Sutton, James. (date)
Power programming the Commodore 64.

Includes index.
1. Commodore 64 (Computer)—Programming.
2. Assembler language (Computer program language)
3. Computer graphics. 4. Computer sound processing.
I. Title.
QA76.8.C64S89 1985 001.64'2 85-3537
ISBN 0-13-687849-0

Editorial/production supervision: *Karen Skrable Fortgang*
Cover photo: *Geoff Gove, The Image Bank*
Cover design: *Photo Plus Art*
Manufacturing buyer: *Gordon Osbourne*

Commodore 64 is a registered trademark of **Commodore Electronics Limited.**

© 1985 by **Prentice-Hall, Inc.,** Englewood Cliffs, New Jersey 07632

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-687849-0 01

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

CONTENTS

JAMES SUTTON

POWER PROGRAMMING THE COMMODORE 64: ASSEMBLY LANGUAGE, GRAPHICS, AND SOUND

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Sutton, James. (date)
Power programming the Commodore 64.

Includes index.
1. Commodore 64 (Computer)—Programming.
2. Assembler language (Computer program language)
3. Computer graphics. 4. Computer sound processing.
I. Title.
QA76.8.C64S89 1985 001.64'2 85-3537
ISBN 0-13-687849-0

Editorial/production supervision: *Karen Skrable Fortgang*
Cover photo: *Geoff Gove, The Image Bank*
Cover design: *Photo Plus Art*
Manufacturing buyer: *Gordon Osbourne*

Commodore 64 is a registered trademark of **Commodore Electronics Limited.**

© 1985 by **Prentice-Hall, Inc.,** Englewood Cliffs, New Jersey 07632

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-687849-0 01

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

CONTENTS

PREFACE: *EXPOSÉ*

The one, central problem we have all faced with computers is how to make our electronic serfs do significant work for us. This question causes difficulties for everyone from the casual user to the serious apprentice or professional programmer.

Those that intend to solve the problem by writing their own programs set off on a quest for "how-to" knowledge. The first leg of this search for the holy grail is spent groping just to discover what subjects to study! Typically, computer information is artificially separated into such topics as assembly language, hardware, graphics, structured programming, and so on, which are then discussed in their own volumes. The circular reasoning involved requires that the student know the subject to be able to study it! The sales of computer books prove that many people are spending hundreds of dollars attempting to gain an integrated understanding of computers; the number of computers being set aside shortly after purchase prove that many people are failing in these attempts. Together, these statistics suggest where the problem lies.

A single book has been needed to guide the computer user from first principles to the ultimate goal of programming, the conversion of vague ideas for computer tasks into specific and complete computer tasks. This view of programming is interdisciplinary at heart and requires that such varied fields as software development, computer resources (chips, speed, etc.), problem solving, and the arts be considered in unison. Filling these complementary needs is the purpose of this book.

Each chapter explains concepts relevant to any computer language, including BASIC, and applies these concepts to the Commodore 64 wherever possible. Much of the computer-specific material is unavailable anywhere else, being gathered during long discussions with practicing Commodore 64 programmers. Almost all the

information can improve your BASIC programming, but as we shall see shortly, the greatest benefit will come from applying these concepts to assembly language programming. For this reason assembly language will be taught and used throughout, with occasional references to BASIC. Each chapter ends with a list of one or two of the best books for deeper study in each subject covered.

One fact has had a major influence on this book: Software, even software written by professional programmers on contract to outside customers, has historically had a dismal success record. Incredibly, only 2% of all programs that have been contracted for have run correctly and have accurately performed the buyer's intended task as delivered! Professionals are supposed to have techniques beyond those available to the home computerist. Some do, but far too many have continued to use a "BASIC User's Manual" approach to programming.

The most important reason for this is the exclusion in most computer tutorials of the methods or even the need for the most basic programming activity, getting a clear picture of an intended computer task *before* writing a program to perform it. Without a method for getting a clear picture of the task, there is no place for a method that carries the clear picture through to a program. Thus most programs have no direct connection with the initial task they are attempting to perform.

The record shows the sorry result for pros and amateurs alike; programs that either do not work at all, or perform the wrong task, or require repeated correcting before they are "acceptable." The latter can be roughly translated as "it's not exactly what I wanted, but I'm tired of bothering with it."

We will unfold a single method that leads you to a thorough definition of your intended task, then to an effective method for performing the task, and finally to an assembly language program that fully executes the original task. Only the last step is language specific; a program can be written in BASIC just as easily as in assembly language from the results of the first two steps.

Once the means for turning a task idea into a program are understood, we will present tools for self-expression. A great part of the satisfaction of programming comes from using your entire creative ability, right and left brain, artistic and analytical. Graphics, sound, words, and plot are the programmer's main artistic points of contact with his audience. Lessons from cinematography and music composition, with other artistic considerations, are discussed in the chapters on graphics and music/sound programming techniques.

Similar power programming books are planned for each of the most popular educational and home computers. A follow-on book entitled *Beyond Power Programming* will provide a reference manual of methods to perform many of the most important advanced programming tasks, including but not limited to; advanced searching and sorting methods, gaming, 3D graphics, artificial intelligence, and programming tools.

Why do we choose to program in assembly language? The answer is found in benchmarks, or comparison tests, of programming languages. One benchmark was conducted on a predecessor of the Commodore 64 and reported in *Creative Com-*

puting magazine.* The task of filling up the TV screen with one character, then filling it with another, and so on, until every character supported by the computer has filled the screen once, was implemented in each language.

Here is the portion of the results dealing with the major languages we have mentioned, adding results for a different high-level language, Pascal, for comparison:

PARTIAL CHART OF LANGUAGE COMPARISONS

Language	Time to run (sec)	Size (bytes)	Speed ratio
PET BASIC	4368.7	302	1.0
Pascal	460.1	556	9.5
Assembly	6.1	105	716.2

The BASIC version of the program took less of author Gregory Yob's time to write and correct than did the other versions, but Mr. Yob attributed much of that difference to his lack of familiarity with assembly language and Pascal. Indeed, small programs like this often run correctly on the first try if written in Pascal. This implies 0 correcting time. Pascal is the fastest and easiest language of the listed three for programming and debugging *large* programs, and is the language of choice wherever BASIC "will do." (Pascal was originally written to teach good programming, and it is worth learning for that reason alone; habits encouraged by BASIC have been shown to make poor programmers.)

Although writing characters to a television screen is not a general test of languages, it does relate to a distinguishing feature of the Commodore 64, the graphics character mode. This mode allows the creation of your own specialized sets of graphics or other symbols for use in programs.

Note that while Pascal is much faster than BASIC, assembly is outrageously faster than either. Your Commodore 64 has the ability in assembly language to display graphics 700 times faster than BASIC allows. This opens up built-in graphics capabilities that are completely unavailable to BASIC programs! For example, in assembly language there is enough time to split the TV screen into horizontal pieces and treat each piece as a separate screen. Each screen can have different features, an option precluded by BASIC's slowness. Assembly language even leaves enough time for a program to do other work "between the cracks" as it controls the visual display. There is more on such graphics techniques in Chapter 6.

A second comparison comes from *Byte* magazine.† This time the task is "The

*Gregory Yob, "Personal Electronic Transactions," *Creative Computing,* November 1982, p. 294.

†Jim Gilbreath and Gary Gilbreath, "Eratosthenes Revisited: Once More Through the Sieve," *Byte,* January 1983, p. 283.

Sieve of Eratosthenes," which finds all the prime numbers up to a given number (in this case, 8191). All the multiples of 2, 3, and of the primes (as they are discovered) are crossed off a number list. When the task is completed, only the primes are left. This test addition (the multiplications are implemented as repeated additions), subtraction (number comparisons), and several other mathematically oriented functions that are common in computer operation.

In this case the languages span several different computers, but each computer mentioned contains the same "brain," or microprocessor, as that used in the Commodore 64, running at the same speed. The results are as follows:

Language	Time to run	Speed ratio
Apple BASIC	1850 sec (31 min)	1.0
Apple Pascal	160 sec (6+ min)	11.7
Assembly (OSI Superboard)	13.9 sec (¼ min)	133.1

You could say that the BASIC version leaves time to finish a full-course dinner, but assembly just barely lets you tie your shoelaces.

There is a place for BASIC programming, but that place does not include high-speed computation, sophisticated graphics, or applications requiring flexible use of the computer's resources. Having been invented in 1965, a long time ago for any computer language, BASIC should be spared harsh criticism. It is, and always was, intended as an introduction to computers.

For all of the preceding reasons, assembly is the language we have chosen to complement a powerful and proven method for accomplishing your computer goals. This welds the lever of program-planning technique with the hammer of assembly language into a force-multiplying tool for Power Programming.

ACKNOWLEDGMENTS

I have many people to thank for their many different kinds of help. For technical help I owe the most to Larry Holland of HES Corp. He is a true gentleman and a Commodore 64 professional. Many of the I/O coding techniques were suggested by him and have been used with telling effect in his own programs. HES Corp. deserves similar thanks for making Larry available to me. Another HES expert who has provided help is Jay Stevens. Any errors are the fault of this author, not of Larry or Jay.

Further, my employer, Lockheed Missiles and Space Company of Sunnyvale, California, has directly supported this book with shifted work hours, and indirectly supported it with an outstanding technical environment to work and grow in. I could not have written as much or as well without them.

For authorial support I owe the most to Bernard Goodwin, my publisher at Prentice-Hall. His patience with my perfectionism and the resulting broken deadlines allowed several extra passes at the book. Another major support came from my brother, whose command of the English language is as unshakable as anyone's I know. His suggestions led to some of the most important changes in the early portions of the text.

For personal support I thank my family: my wife, Debbie, and daughters Noelle, Jenyfer, and Natalie. They sacrificed most of their time with me for nearly two years. Additionally, Debbie acted as a reviewer, and often saw the forest while I was wandering among the trees. Her suggestions greatly improved the organization of a number of sections.

Finally, I thank my Lord Jesus for the encouragement his words have given me to see through this twenty-two-month pregnancy with *Power Programming*. Without that encouragement this "baby" would have miscarried on several occasions.

PAST ITS PLASTIC ENVELOPE: THE COMPUTER'S INNER MACHINERY

The London street light edges the window like a mute sunburst. Lying on oak, raggedly stacked pages are swirled by a mantle's glow, but their inner current of symbols and mechanical drawings is diverted only by the occasional footnote "1842 REVISION." The Prime Minister reaches to blacken a quill, then marks the final page "Rejected." Mankind is denied the computer for another hundred years.

The rejected manuscript was the blueprint for a computer built of mechanical parts. Its inventor, Englishman Charles Babbage (1792–1871), alienated fellow scientists by criticizing the bureaucratic leaders of their scientific societies. In return, those leaders lobbied the British government, which then withdrew its financial support for the development of Babbage's computer. The completed blueprint lay forgotten after the inventor's death in 1871. Its rediscovery in 1937 and updated development during World War II resulted in the first electronic digital computer, ENIAC (Electronic Numerical Integrator and Calculator), in 1946.

The mechanical computer, which Babbage called the Analytical Engine, had all but two of the basic attributes of modern computers. These attributes are easily understood in the mechanical computer since the physical actions supporting them are familiar to us from everyday life. We will present these basic attributes first in mechanical terms, and then apply them to the electronic actions and circuitry of your modern computer. The analogy is simple and direct. Even the roots of the two basic modern-computer functions absent in Babbage's invention can be traced to the mechanical computer.

THE "NUMBER MILL"

The heart of the computer was originally called the "mill." Babbage named it after the grain mill, a machine that converts raw grain, or grist, into flour. In a grain mill, grist is placed between two stone disks and one disk is turned, usually by a water-wheel. The trapped grist is pulverized into flour. In both grain mills and computers the raw material and end product are made of the same substance; only the form changes. The substance milled by a computer is not, of course, grain. It is numbers. Numbers entering the mill will be "mashed," that is, added, subtracted, and otherwise transformed, and number results of a different form exit.

Number transformation is the central activity of the computer. Outside the mill portion of the computer is a miniature world centered around and supporting the mill. This world has storage bins for the grist, a transportation system to move grist between mill, storage, and consumers, and a communications system to coordinate all these activities. "Consumers" are often devices that show human beings the milling results.

All these milling support systems are present in modern computers. Two other attributes common to both the Analytical Engine and today's computers are grist made of binary instead of decimal numbers, and a program, or sequence of instructions like a recipe, to direct the mill's operations.

All mathematical machines prior to the Analytical Engine, including an earlier calculator invented by Babbage, used the decimal or base 10 number system for calculations. The decimal system, based on our two five-fingered hands for a total of 10 "digits," complicates machinery by requiring machine representations for 10 different numbers from 0 to 9. Its rules for arithmetic are convoluted, as seen in the rules for carrying or borrowing a 1. On the other hand, the binary system of two digits, 0 and 1, has the very fewest number of digits that can convey a useful meaning: With two digits one can both express numbers and modify them with simple arithmetic operations.

Just *two* different mechanical or electronic digit representations are needed in a binary-based machine. This not only simplifies computers, it also structures the computer machinery more regularly. This structuring makes construction of computers more practical.

The original concept of a program was at once radical and elementary. To Babbage, a program was more than just a sequence of number transformations for the mill to perform on grist; it implied the ability of the mill to look at the result of a transformation and then *choose a next transformation!* This was the beginning of the modern computer program.

For example, a program may direct the following mill actions in response to these different results of a subtraction: If the result is negative, move a storage location's contents; if positive, do another subtraction; and if zero, "wave a (figurative) flag" to alert the human being operating the computer. The ability to select among alternative actions allows the mill to skip unnecessary commands, repeat parts of a program, or pick one of many paths through the program.

When the mill completes the commands in a nontrivial program, numbers have been moved and transformed to complete a desired task. Turning statistical facts into actuarial tables for the insurance industry was one such task envisioned by Babbage.

Babbage insisted on another function in his computer: The Analytical Engine had to be able to show the results of each command executed during the running of a program. To do this, a means was developed to connect the mill to devices in the outside world. In this case the outside device recorded the program results for human inspection.

To tell the Babbage mill what transformations to perform, in what order, with what options, and on what numbers, four types of punched cards were used:

Constant cards, holding the actual numbers to be operated on.

Variable cards, specifying the *locations* of grist storage bins whose numerical contents are to be operated on.

Operations cards, detailing the operations to be performed by the mill.

Directive cards, directing the movement of numbers within the Engine, and directing the choice of the program command to be executed next, based on the results of previous operations.

The same four categories of information for the mill are also found in microcomputer programs. Constants and variables are grouped under the name *operands,* because operations can obtain grist from either of them. Again, note this difference: A constant is a *number* to be operated on; a variable is the *address* of a storage bin holding a number to be operated on.

Except for constants, information on the Babbage cards was nonnumerical and could not be stored in the mill's internal and numerical storage. Thus the cards were read by the mill as needed, and the program was said to be externally stored. This limitation was largely responsible for the development of the two most recent basic principles: codes and internal program storage.

A *code* is a meaning assigned to a number. The number 127 could be chosen to represent the subtraction operation, for instance. Then if 127 enters the mill in a special operation code context, the mill will interpret it to mean "perform a subtraction." If entering the mill during another context, 127 might be interpreted as a variable (address of a storage bin) containing number grist. Or again, 127 might simply be taken for a constant to be used as grist. Numbers can now be used not only as the internal grist to be operated on, but also as the representatives of operations and variables.

This opens the door for the second advance: *internal program storage.* With the entire program represented numerically, its parts could now reside in internal storage bins. This allows the modern mill to use the normal transportation system to retrieve "operators" (representing operations) and operands, and to accomplish new goals that are impractical with an externally stored program.

These attributes—a central math mill, binary-number grist, a supporting world of storage and transportation, and an internal program represented by binary numbers—are the bases of modern computer structure (including that of microcomputers). We now know how the principles came about, and more important, why. Each serves a need that develops when one begins designing a machine to perform unrestricted tasks by transforming numbers. To use these attributes we must study them in action.

Since binary numbers are central to both grist and program, we will start with a broader view that allows for grist, program, and much more.

Exercise:

Answer the following question.

(a) What are the four bases of the modern computer?

INFORMATION

From the concept of codes, mentally step up one level of abstraction. You have lighted upon the "super-concept" of information.

Webster's defines *information* as "a logical quantity. . . ."* A *quantity* is something that is mathematically concrete and that is manipulable by mathematical tools. Almost every aspect of human knowledge can be treated mathematically, which is why mathematics is called "the universal language."

When information is assigned to numbers via a code, the result is called *data*. Being numerical, data can be directly transformed through mathematics, whereas information usually cannot be transformed so directly. Thus, information is transformed by coding it as data, applying mathematical operations to the data, and decoding the resulting data into new information. External devices such as keyboards and video circuits often code and decode information, while the computer itself transforms the data.

An example of this cycle occurs with alphabetic input from a keyboard. A human being types in a message, which the keyboard circuitry converts into coded numbers or data and supplies to the computer. The computer transforms the data under the direction of a program and supplies the results in numerical form to video circuitry. The video circuitry converts the numbers into a form that a television or video monitor can display as characters readable by a human being. Since the alphabet is sequential, the computer can transform an A into a B by adding 1 to the data value representing the A, and so on up the alphabet.

With this broadening of our perspective the number mill becomes the information mill, bringing words, pictures, sounds, and ideas under the control of the pro-

* *Webster's Third New International Dictionary,* G. & C. Merriam Company, Springfield, Mass., 1971.

grammer's imagination. The pioneer who exploits codes creatively will be placing an uncharted region into his domain.

But as quickly as we have entered that mystic realm, we must retreat to the known colonies of existing codes. Their mastery is essential preparation for any further sorties.

Three categories of codes warrant our attention: the numeric, the symbolic, and the conceptual. The numeric codes are foundational and will require the most effort to master. Your time with these codes will be repaid many times over in Chapter 3 when we study their transformation with assembly language.

Numeric Codes

The *numeric codes* assign number meanings to the numbers used within the computer. Numerically coded numbers can be used in counting and all other mathematical activities. They can also indirectly represent anything that mathematics can manipulate; that is, they can indirectly represent any information. So numerically coded numbers can represent the number of shopping days left until Christmas, the number of times the computer must perform a particular set of actions, and so on. This is analogous to the role of numbers in equations. Numbers are treated by equations as pure values, but may and often do represent physical or conceptual quantities to the human being utilizing the equations.

The most general codes are numeric. Since these codes can indirectly represent any information, there may seem to be little need for other types of codes. However, the remaining code types restrict and predefine the assignment of information to numbers in certain special cases to simplify and enlighten the programmer's job.

We will examine three numeric codes: the binary code, the hexadecimal code, and the binary-coded decimal (BCD) code. Be sure that you understand each code as it is discussed before moving along; the explanation is graduated and depends on your grasping all earlier material.

Binary code. The *binary code,* also called the *binary number system,* represents arbitrarily large or small numbers with combinations of the digits 0 and 1. Binary number representation is similar to that of the decimal system, but simpler.

Imagine a row of light switches on a wall. Each switch has the standard Off and On positions. Let's say that these represent the binary digits 0 and 1, respectively (Fig. 1.1). Using the rightmost switch alone allows for only two number values, 0 and 1 (Off and On). Using the two rightmost switches allows for four number values; two for the Off and On positions of the rightmost switch while the next-left switch is Off, producing 00 and 01, and two for the same positions of the rightmost switch while the next-left switch is On, producing 10 and 11.

Each added switch doubles the total number of combinations, since all previous combinations are allowable in combination with each of the two positions of the new switch. Saying this with numbers, we have 2 (or 2^1) possible position com-

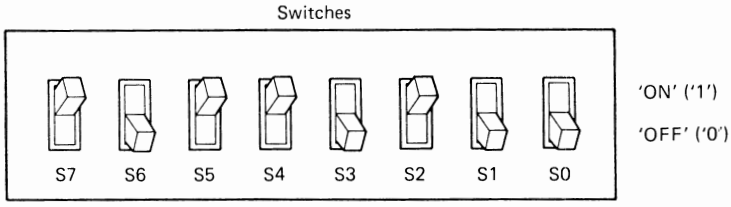


Figure 1.1

binations with one switch, 4 (two times 2 or 2^2) combinations with two switches, 8 (two times 4 or 2^3) combinations with three, and in general, 2^n combinations with n switches. The i^j notation means "multiply i by itself j times," as in $2^3 = 2 \times 2 \times 2 = 8$. In the base 10 or decimal system, the rule would be 10^n combinations with n 10-position switches.

Each two-position switch in the wall plate corresponds to one column or *bit,* short for "BInary digiT," in a binary number. Bits are grouped together, like the switches above, to express more numbers than would be possible with a single bit. Early computers actually used On/Off switches like those above for entering data.

All data expressed as a sequence of On-Off bits are called *digital data,* due to the grouping of digits. Eight-bit numbers are the norm in microcomputers like the Commodore 64, and are called *bytes.* We now know that an 8-bit byte allows $2^8 = 256$ different bit combinations.

In our discussions of binary numbers we shall use and assume byte groupings of bits. Four-bit *nibbles* and 16-bit numbers are also common. Any bit grouping, regardless of the number of bits involved, is generically called a *word.* The entire switch/bit analogy can be demonstrated visually as shown in Fig. 1.2. This analogy is especially close since bits are represented in the microcomputer by tiny switches. These on/off devices are familiar to you as *transistors.* They provide a *physical* way to represent numbers. The remainder of our discussion of binary numbers will concentrate on binary math and how special types of numbers can be represented in the binary code.

Binary counting begins with the number zero, which is represented by a binary word containing only 0-bits. We say that all the bits have been *reset,* since "setting" a bit means making it a 1. Thus the byte value 0000 0000 equals the number zero. The digits are grouped in fours to make reading them easier.

Beginning as in the decimal code, the rightmost column of a 0-byte increments to 1. Then, since there are only two possible digits per column, a succeeding incre-

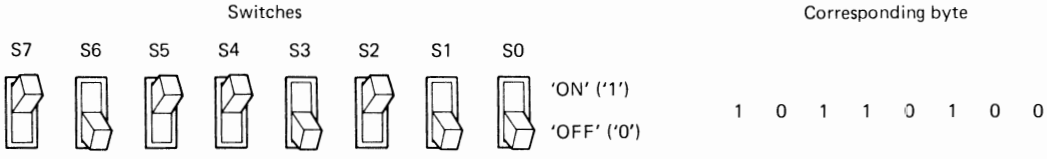


Figure 1.2

ment returns the rightmost bit to 0, and a *carry* is made to the next bit to the left. So it goes, incrementing and carrying, as shown in Table 1.1.

Thus a 1 value in the rightmost binary column and 0's in all others represents a decimal value of 1. A 1 in the column left of rightmost and 0's elsewhere represents the decimal value 2. Moving over one more column, a lone 1 digit represents decimal 4. Following this pattern, the *weight* or multiplier for a 1-bit in each column is a power of 2. 2^0, 2^1, and 2^2 are the weights for the three rightmost columns just discussed. Put a little more formally, the columns in a byte are numbered from 0 to 7 and are labeled d7 d6 d5 d4 d3 d2 d1 d0. The d0 or rightmost bit has a weight of 2^0, and the d7 or leftmost bit has a weight of 2^7.

The first 16 powers of 2 are used quite frequently. They are listed in Table 1.2. Note that the d7 bit of an 8-bit number has a weight of 2^7 or 128, while a byte as a whole can represent 2^8 or 256 possible numbers. This is because the number resulting from setting the d7 bit is the binary value 1000 0000, or decimal 128, while a byte as a whole can represent 256 different *numbers* from 0000 0000 to 1111 1111.

One last point about the table of powers: 2^{10} has a special place in binary terminology. Its decimal equivalent, 1024, is close to a convenient power of 10, the number 1000. In electronics and other fields, the number 1000 is often abbreviated "K". Because of its kinship to the number system we know and love, we tend to speak of large binary numbers in terms of K. So $2^{16} = 2^6 \times 2^{10} = (2^6)K = 64K$, and so on.

Binary-to-Decimal Conversion. To find the decimal equivalent for a binary number, just add the weights of all columns containing 1's. If you like, you can also add in the columns containing 0's as place-markers.

TABLE 1.1 BINARY-DECIMAL EQUIVALENTS

Binary value	Decimal value
0000 0000	0
0000 0001	1
0000 0010	2
0000 0011	3
0000 0100	4
0000 0101	5
0000 0110	6
0000 0111	7
0000 1000	8
.	.
.	.
.	.
1111 1111	255

TABLE 1.2 POWERS OF 2

Power of 2	Decimal value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1,024
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536

Example:

Convert the binary number 0110 1011 to its decimal equivalent.

$$0 \times (2^7) + 1 \times (2^6) + 1 \times (2^5) + 0 \times (2^4) + 1 \times (2^3) + 0 \times (2^2) + 1 \times (2^1) + 1 \times (2^0)$$
$$= 0 + 64 + 32 + 0 + 8 + 0 + 2 + 1$$
$$= 107 \text{ decimal.}$$

Exercise:

Try the following conversions. The answers are given in Appendix A.

(b) 1001 0000 =

(c) 0010 1111 =

(d) 1101 0111 =

Decimal-to-Binary Conversion. To find the binary value of a decimal number, reverse the conversion process of the preceding section. First divide the decimal number by the largest power of 2 that still leaves a remainder. Then divide the remainder by the next-lowest power. Continue dividing and isolating a remainder until you have made the last division, a division by 2^0 (= 1).

Example:

Reverse the preceding example and calculate the binary equivalent of 107 decimal.

107 decimal can be divided by 2^6 (64) but not by 2^7 (128). So

$107/2^7 = 0$ remainder 107
$107/2^6 = 1$ remainder 43
$43/2^5 = 1$ remainder 11
$11/2^4 = 0$ remainder 11
$11/2^3 = 1$ remainder 3
$3/2^2 = 0$ remainder 3
$3/2^1 = 1$ remainder 1
$1/2^0 = 1$ remainder 0

The binary result by weighted column appears as follows:

d7	d6	d5	d4	d3	d2	d1	d0
0	1	1	0	1	0	1	1

as before.

Exercise:

Here are a few decimal-to-binary conversions with which to practice.

(e) 253 =

(f) 52 =

(g) 147 =

You are already close to having a working knowledge of binary numbers. To complete your understanding, we must tidy up the details of binary addition, subtraction, multiplication, division, and the shift and logical operations.

Addition. The following four operations are all that are needed for binary addition.

```
  0        0        1        + 1      1   (e.g., a carry)
+ 0      + 1      + 1        + 1
---      ---      ---        ---
  0        1       (1)0      (1)1
                  (0 carry 1) (1 carry 1)
```

Each column in a binary addition of two numbers will contain one of these four operations.

Example:

Adding the byte values 1011 1101 and 1001 1100 demonstrates all the addition operations in one problem. Each column addition is visually separated on different levels as it is performed below.

```
  1 0 1 1 1 1 0 1
+ 1 0 0 1 1 1 0 0
-----------------
                1
              + 0
              ---
                1
              0
            + 0
            ---
              0
            1
          + 1
          ---
            1 0
        + 1
        + 1
        ---
          1 1
      + 1
      + 1
      ---
        1 1
    + 1
    + 0
    ---
      1 0
  + 0
  + 0
  ---
    1
  1
+ 1
---
1 0
-----------------
1 0 1 0 1 1 0 0 1
```

Both addends in the preceding example were 8-bit bytes, but the sum was 9 bits long, or 1 bit larger. In a byte-wide computer such as the Commodore 64, this extra bit is called the *carry bit* or *carry flag*. It is outside the storage location for the result byte, but its value is stored elsewhere so it can be checked by the program.

Exercise:

Practice the binary operations with the following addition problems.

(h)
```
  0 1 1 1 1 1 1 1
+ 0 1 1 1 1 1 1 1
```

(i)
```
  1 0 1 0 1 0 1 1
+ 1 1 1 1 1 1 1 1
```

(j)
```
  1 1 1 0 0 1 0 1
+ 1 0 1 0 0 1 1 0
```

Subtraction. True binary subtraction is a rarity because the binary code makes it easier to convert a subtraction problem into an addition problem and solve the addition than to solve a subtraction problem as is. So, given a number *X*, a number *Y*, and the subtraction problem $X - Y$, the normal solution approach is to find the negative of *Y* and to solve the equivalent problem $X + (-Y)$.

This approach requires that the binary code be modified to allow for negative numbers. One way to express a negative number in binary is to set aside the d7 bit of a byte as the sign bit. If d7 is equal to 0, the value of the other 7 bits is interpreted as positive. If d7 equals 1, the value of the other 7 bits is negative. Thus 1000 0001 = −1 decimal, and 0000 0001 = 1 decimal. This coding format is called *signed binary*. Signed binary is a variation of the binary code that is simple to understand and that has a specific application of importance that will be discussed shortly. In general, though, signed binary numbers are less well suited to subtraction than another scheme, which we discuss next.

A value called the *arithmetic negative* can be calculated with little effort from any binary number. Arithmetic negatives are important because their use allows numbers to be added and subtracted by one set of mill circuits. This aids the twin goals of keeping the computer and its arithmetic as simple as possible.

There are two ways to obtain the arithmetic negative of a binary number. One is easily understood but lengthy in use, and the other is a shortcut. Let's walk through the long way once to help us to understand how the shortcut works.

Recall that the number $-Y$ is equivalent to $0 - Y$. In the long method we will start by subtracting the binary number to be negated from the number 0.

Since the computer uses only 8 bits to represent numbers, it treats a byte of 8 0-bits as a zero, regardless of whether or not there is a ninth (carried) 1-bit. We use this fact to help set up the subtraction. Let's attach a ninth 1-bit, a d8, to the 8 0-bits

from which we are subtracting. The computer does not care, and it gives us a bit to borrow from. Thus the subtraction will be of the form

$$1\ 0000\ 0000 - Y$$

As in addition, there are four possible column operations. Each column subtraction looks like one of the following:

1	1	0	10	(this is a 0 − 1 after
− 1	− 0	− 0	− 1	borrowing from the
0	1	0	1	ninth 1-bit)

We will illustrate the initial $0 - Y$ subtraction of the long method in an example problem containing all four subtraction operations.

Example:

Negate Y, for $Y = 0110\ 1110$.

As we have just said, the negative of a number can be found by subtracting it from 0. With the ninth bit set to provide a source to borrow from, this problem appears as follows:

1 0 0 0 0 0 0 0 0	Stage 1	(0 − Y; restate this problem with the
− 0 1 1 0 1 1 1 0		carry bit borrowed down the byte)
1 1 1 1 1 1 10 0	Stage 2	(the carry bit has been borrowed down
− 0 1 1 0 1 1 1 0		to the last negative-result column)
1 0 0 1 0 0 1 0	Stage 3	(− Y)

so the arithmetic negative of −01101110 is 10010010.

A full subtraction problem using this means of negation can be expressed as

$$X - Y = X + (1\ 0000\ 0000 - Y)$$

With the additional fact that

$$1\ 0000\ 0000 = 1111\ 1111 + 1$$

we say

$$X - Y = X + (1\ 0000\ 0000 - Y) = X + ([1111\ 1111 - Y] + 1)$$

The term in brackets, $1111\ 1111 - Y$, holds the key to the shortcut.

Example:

Observe the results of the operation $1111\ 1111 - Y = Z$ on a few values for Y:

(i)
1 1 1 1 1 1 1 1	
− 0 1 1 0 1 0 1 1	Y
1 0 0 1 0 1 0 0	Z

(ii)
$$\begin{array}{rr} & 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ -\ & 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1 \quad Y \\ \hline & 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \quad Z \end{array}$$

(iii)
$$\begin{array}{rr} & 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ -\ & 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1 \quad Y \\ \hline & 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0 \quad Z \end{array}$$

Each bit in each result Z is reversed from the corresponding bit in Y. Thus, instead of subtracting Y from 1111 1111, each bit in Y can simply be reversed. The reverse of Y is called the *logical NOT* of Y, since when it is complete every bit in Y is NOT the same as in the original.

Thus the shortcut reduces to the equation

$$X - Y = X + ([\text{NOT } Y] + 1)$$

In words, this equation says "To subtract Y from X, reverse every bit in Y and add 1. Then add the result to X." The subtraction has been converted to an addition, and the promised simplicity has at last arrived.

Example:

Calculate $X - Y$ for $X = 1100\ 1011$ and $Y = 1001\ 0110$.

First, reverse each bit in Y:

$$(\text{NOT } Y) = 01101001$$

Next, add 1:

$$\begin{array}{rr} & 01101001 \quad = (\text{NOT } Y) \\ + & \quad\quad 1 \\ \hline & 01101010 \quad = (\text{NOT } Y + 1) \end{array}$$

Finally, add (NOT Y + 1) and X:

$$\begin{array}{rl} & 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1 \quad X \\ + & 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0 \quad (\text{NOT } Y + 1) \\ \hline (1) & 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1 = X - Y \end{array}$$

(we discard the ninth bit since it is not used)

As a double-check, convert X, Y, and our value for $X - Y$ into decimal:

$$\begin{array}{rrrrrrrrrr} X & = & 128 & + & 64 & + & 8 & + & 2 & +1 & = & 203 \\ Y & = & 128 & + & 16 & + & 4 & + & 2 & & = & 150 \\ X - Y & = & 32 & + & 16 & + & 4 & + & 1 & & = & 53 \end{array}$$

It checks.

Of course, the foregoing method also works with longer data words, as in the following example.

Example:

Solve $X - Y$ for 16-bit words where X = 10010111 11101101 and Y = 00010101 01101001.

Following the shortcut, we find that

$$(\text{NOT } Y) = 11101010\ 10010110$$

and

$$(\text{NOT } Y) + 1 = 11101010\ 10010111$$

so

$$\begin{array}{rr} X + [(\text{NOT } Y) + 1] = & 10010111\ 11101101 \\ + & 11101010\ 10010111 \\ \hline & 1\ 10000010\ 10000100 \end{array}$$

The carry, or seventeenth (d16) bit, is discarded, so the result is

$$X - Y = 10000010\ 10000100$$

Exercise:

Hone your subtraction skills by solving the following problems. Do (k) as a long subtraction, and (l) and (m) as additions using the conversion process described above.

$$\begin{array}{ll} \text{(k)} & 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ - & 0\ 1\ 1\ 1\ 1\ 1\ 0\ 1 \\ \hline \text{(l)} & 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1 \\ - & 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1 \\ \hline \text{(m)} & 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0 \\ - & 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1 \\ \hline \end{array}$$

Another term for NOT Y is the *one's complement* of Y, due to its origin by subtraction from a number consisting of all 1's. This is usually abbreviated 1C(Y). NOT $Y + 1$ is an even more important value called the *two's complement* of Y. It is abbreviated 2C(Y).

As in signed binary, the top bit in two's-complement numbers equals 0 for positive numbers and 1 for negatives. Practically speaking, this means that 8 bits will not count to +256 in two's-complement notation. Instead, an 8-bit 2C number represents the integers from 0 to 127 and from −1 to −128. With 16-bit words the representation is of all integers from 0 to 32,767 and from −1 to −32,768, with d15 being the sign bit.

If two large positive 2C numbers are added together, the sign (top) bit of the result can be set to 1. The sum will then erroneously be interpreted as being negative. Similarly, the sum of two large negative numbers can misleadingly appear positive when the top bit is reset to 0. Both conditions are known as *overflow*. Overflows can be avoided by switching to larger words (e.g., from 8 bits to 16 bits).

Minute Numbers. This is a good time to show how minute numbers, or fractions, can be represented. Recall that each column has a weight or power of 2

associated with it. The rightmost column has the weight 2^0, and the power increases incrementally as one goes column by column to the left.

We did not mention what would happen if we were to move to the right starting at the 2^0 column. Bit columns can be placed in these positions with the power of their weight decrementing for each position moved rightward. A "binary point" is placed immediately to the right of the 2^0 column, just as a decimal point is placed to the right of the 10^0 column in base 10. By weights, the columns on both sides of the binary point would appear as

$$\ldots 2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0\ .\ 2^{-1}\ 2^{-2}\ 2^{-3}\ \ldots$$

where the minus sign on the power means "1 over." Thus the multipliers for the first two bit columns right of the binary point are

$$2^{-1} = 1/(2^1) = 1/2 \qquad \text{and} \qquad 2^{-2} = 1/(2^2) = 1/4$$

In this system the number 00010101.01 converts to $16 + 4 + 1 + 1/4 = 21.25$ decimal.

Two's-complement arithmetic with digits to the right of the binary point is simple to perform. Again, given the problem $X - Y$, we must convert to the problem $X + (-Y)$. Instead of having

$$-Y = (\text{NOT } Y) + 1$$

however, we have

$$-Y = (\text{NOT } Y) + .00\ldots1$$

The 1 farthest to the right of the binary point in the .00 . . . 1 term is placed in the same column as the rightmost bit of NOT Y; otherwise, its addition would not affect all NOT Y bits.

Example:

Solve the following subtraction problem:

```
  1 0 1 0 1 0 0 0 . 1 1     minuend X
- 0 1 1 0 0 0 1 1 . 0 1     subtrahend Y
-------------------------
```

First, solve for NOT Y:

$$\text{NOT } 0110\ 0011.01 = 1001\ 1100.10$$

Then find the 2C of Y:

```
  1 0 0 1 1 1 0 0 . 1 0     NOT Y
+               . 0 1     .01 = 1 0000 0000.00 -
-------------------------         1111 1111.11
  1 0 0 1 1 1 0 0 . 1 1     -Y = (NOT Y) + .01
```

Finally, add the minuend to the 2C version of the subtrahend.

```
  1 0 1 0 1 0 0 0 . 1 1
+ 1 0 0 1 1 1 0 0 . 1 1        X + −Y
(1) 0 1 0 0 0 1 0 1 . 1 0      (drop the ninth bit)
```

A decimal check confirms the result: 168.75 − 99.25 = 69.5.

Exercise:

Solve the following problems using any method you like.

(n)
```
  1 0 0 1 1 0 0 1 . 0 1 1
− 0 1 1 1 0 0 1 1 . 1 0 1
```

(o) Convert the 2C binary number 1011.1101 into decimal form.

Multiplication. To multiply two decimal numbers together, we first obtain the partial products due to multiplication by each digit in the multiplier, and then sum them.

Example:

Multiply the decimal numbers 243 and 768.

```
  243
× 768
 1944
 1458      partial products
1701
------
186624     product
```

Similarly, to multiply two binary numbers, we obtain the partial products due to each digit in the multiplier and sum them. Binary column multiplication, like binary column addition and subtraction, is based on four operations. They are

```
  0       0       1       1
× 0     × 1     × 0     × 1
---     ---     ---     ---
  0       0       0       1
```

The rules for binary-point placement in binary multiplication are the same as those for decimal-point placement in decimal multiplication. An example of binary multiplication that incorporates all four types of column multiplications with fractional binary numbers is given below. It illustrates the foregoing principles and completes our discussion of multiplication for now. This subject will be taken up again in greater detail when we study assembly language in Chapter 3.

Example:

Multiply the binary numbers 101101.01 and 1.01.

```
   101101.01
 ×      1.01
 ------------
   10110101
  00000000
 10110101
 ------------
 111000.1001
```

Division. Binary division requires subtracting the largest possible multiple of the divisor from the dividend, doing the same to the remainder from the first subtraction, and so on, until there is an exact result or a satisfactory approximation of the exact result is attained. You might stop with the latter if the exact result contains an infinite number of digits, for example.

Example:

Divide 10011.01 by 11.1.

```
                    101.1
divisor  11.1   ) 10011.01   dividend
                - 111
                  -----
                    101.01
                  - 000
                  -------
                    101.01
                  - 11.1
                  -------
                      1.11
                  - 1.11
                  ------
                      0.00
```

Division will be discussed again in Chapter 3.

Exercise:

Practice binary multiplication and division with the following problems.

(p)
```
   1 1 0 1 . 0 1
 × 1 1 0 1 . 1 1
 ---------------
```

(q) $1\ 0\ 0\ .\ 1\ \overline{)\ 1\ 1\ 1\ 0\ 1\ 1\ .\ 1\ 0\ 1}$

There is another mathematical operation that is unfamiliar to many people, but that aids in the programming of multiplication and division while having many other uses of its own. It is called the shift.

Shifts. A *shift* operation moves every bit of a number to the left or right one column-place for a "shift left" or a "shift right," respectively. Since column

weights double going left, and halve going right, moving a digit one column either doubles or halves its multiplier. The whole number is the sum of these digits times their weights, so shifting the number left multiplies it by 2, and shifting it right divides it by 2.

This allows us to perform a nice trick for decimal-to-binary conversion. If we divide a decimal whole number by 2, we are in effect shifting it as a binary number to the right. A remainder of 0 means that the value 0/2 passed the decimal point and the equivalent binary point into the first fractional column. A remainder of 1 means that 1/2 ended up in the first fractional column. The value remaining to the left of the decimal point is the same as if the original number's binary equivalent had been shifted right one place.

If we collect those remainders until nothing remains left of the decimal point, we will have obtained the entire binary equivalent. The best way to convey this process is with an example.

Example:

Convert the decimal number 55 to its binary equivalent.

Divide the number 55 by the number 2 until the results are completely fractional:

55/2 =	27	remainder 1	
27/2 =	13	remainder 1	
13/2 =	6	remainder 1	
6/2 =	3	remainder 0	
3/2 =	1	remainder 1	
1/2 =	0	remainder 1	done, since the next division 0/2 shows nothing remaining left of the binary point in the binary equivalent

Thus the binary equivalent of 55 is 111011.

This trick works equally well for converting fractional decimal numbers to their binary equivalents. However, fractional numbers are multiplied rather than divided by 2 to shift the binary equivalent's digits left of the binary point. As each digit crosses the decimal point it is collected and removed from the fractional part to be multiplied in the next step.

Example:

Convert the decimal number .671875 to its binary equivalent.

Multiply the number .671875 by 2 until the results are completely nonfractional.

2 × .671875	= 1.34375	
2 × .34375	= 0.6875	
2 × .6875	= 1.375	
2 × .375	= 0.75	
2 × .75	= 1.5	
2 × .5	= 1.0	done, since 0 remains right of the decimal (and binary) point

Thus the binary equivalent of .671875 is .101011.

Decimal numbers containing both whole and fractional parts can be converted by treating the parts separately.

A variation of this trick makes converting from rational (whole-number) fractions even easier than converting from decimal-point fractional numbers. Again we multiply by 2, shifting the equivalent's bits to the left of the binary point.

Example:

Convert the rational fraction 1/3 to its binary equivalent.

As before, multiply by 2 until a nonfractional result is obtained:

$$2 \times 1/3 = 0\ 2/3$$
$$2 \times 2/3 = 1\ 1/3$$
$$2 \times 1/3 = 0\ 2/3$$
$$2 \times 2/3 = 1\ 1/3$$
etc.

Since no nonfractional result will ever be obtained, and since the multiplication results form a pattern, the binary equivalent has an infinite repeating pattern:

$$1/3 \text{ decimal} = .010101 \ldots \text{ binary}$$

Most conversions are only approximate, and must be limited to avoid infinitely long binary results.

Exercise

Try the following problems to ensure that you will remember the conversion trick.

(r) Convert 72 decimal to binary.

(s) Convert .78519 to an eight-place approximate binary number.

(t) Convert 5/6 to an eight-place approximate binary number.

Logical Operations. The shift is sometimes loosely grouped with the *logical operations,* another development of the nineteenth century. In 1859, George Boole (1815–1864), an Englishman like Babbage, originated the theory and operations of mechanized logic. This new form of mathematics came to be called *Boolean Algebra.* Boole's work had a warmer reception than Babbage's, but its implications to automated computation were unrecognized until 1938.

There are four major logical operations. We already know about one of them, the NOT operation, from our earlier discussion of subtraction. Recall that the NOT operation reverses, or *complements,* every bit in a number. So NOT 1000 1000 = 0111 0111, and so on.

Next is the *OR* operation. Like the other binary operations we have studied, it takes two input values and produces a single result. In logic theory, the 0's and 1's that we have been using are equated with another duality, that of False and True, respectively. Logic, like philosophy, revolves around finding truth, so if one term OR the other is true, the result is True. In fact, any OR operation containing a True argument has a True result. A *truth table* of the OR function, showing it in terms of False and True, or 0's and 1's, illustrate these relationships (see Table 1.3).

TABLE 1.3 OR TRUTH TABLE

OR (Logical)			OR (Binary)		
Inputs			Inputs		
A	B	Result	A	B	Result
False	False	False	0	0	0
False	True	True	0	1	1
True	False	True	1	0	1
True	True	True	1	1	1

To OR two bytes or words together, OR their corresponding columns one at a time (i.e., *bitwise*).

Example:

Logically OR the values 0010 0001 and 1000 1101.

```
   00100001
OR 10001101
   --------
   10101101
```

A common use of the OR operation is to apply a "conditioning byte" to set certain bits in a target byte to 1. For instance, the conditioning byte 1000 0000, ORed with any other byte, will ensure that the other byte's d7 bit equals a 1.

Third is the AND function. It requires a True AND a True to produce a True. The truth table appears in Table 1.4.

Binary bytes or words are ANDed together bitwise.

Example:

Logically AND the values 0010 0001 and 1000 1101.

```
   00100001
AND 10001101
   --------
   00000001
```

TABLE 1.4 AND TRUTH TABLE

AND (Logical)			AND (Binary)		
Inputs			Inputs		
A	B	Result	A	B	Result
False	False	False	0	0	0
False	True	False	0	1	0
True	False	False	1	0	0
True	True	True	1	1	1

A 1-bit in an AND operation is like a gate; it passes whatever value is in the corresponding bit of the other value. That is,

Gate	Operation	Target		Target result
1	AND	0	=	0
1	AND	1	=	1

On the other hand, a 0-bit in an AND operation "masks" out the other bit, regardless of its value, and gives a 0 result. That is,

Gate	Operation	Target		Target result
0	AND	0	=	0
0	AND	1	=	0

One use of the AND function is in testing the contents of bits in a byte or word. To discover if bit d5 of a target byte holds a 1, AND the byte with a conditioning byte of 0010 0000. The operation zeroes all bits but d5, gates d5, and produces a directly testable nonzero-result if d5 originally contained a 1.

The last logical operation, *XOR* or *Exclusive OR,* produces a True result from two inputs when EXCLUSIVELY one OR the other, but not both, is True (see Table 1.5).

The XOR operation can be used to set a byte equal to 0. Simply XOR the byte with itself.

Example:

Logically XOR the value 1001 1011 with itself.

```
          10011011
XOR       10011011
          --------
          00000000
```

Another use comes from the property that a 0 XORed with any bit produces the original value of the bit, but a 1 XORed with any bit produces the reverse of the original bit value. Thus selected bits in a byte can be complemented.

TABLE 1.5 XOR TRUTH TABLE

XOR (Logical)			XOR (Binary)		
Inputs			Inputs		
A	B	Result	A	B	Result
False	False	False	0	0	0
False	True	True	0	1	1
True	False	True	1	0	1
True	True	False	1	1	0

The symbols for NOT, OR, AND, and XOR are the overbar ($\bar{}$), plus (+), dot (•), and circled plus (⊕), respectively.

Exercise:

Perform the following logical operations.

(u) 1 0 1 1 0 0 1 1
OR 0 0 1 0 1 1 1 0

(v) 0 1 0 0 1 0 1 0
AND 1 0 0 1 1 1 1 0

(w) 1 0 1 1 0 1 1 0
XOR 0 1 1 0 1 1 0 0

Hexadecimal code. *Hexadecimal code,* often simply called *hex,* is a numeric code used as a shorthand method for notating binary numbers; it is a programmer's tool. Nevertheless, it is also a legitimate number system in the same sense as the binary code. We will consider hex from both of these perspectives: as a binary shorthand and as an independent number system.

Code Conversions. When discussing the binary code, we visually divided binary numbers into 4-bit groups for readability (e.g., 1011 1001). Four bits allow for 2^4 or 16 different values. It follows that a number system allowing for 16 possible digit values in each column could replace every binary 4-bit group with a single digit. This would further increase readability. The hexadecimal system fits this definition. Its root words, meaning "six" and "ten," reflect its number base of 16.

The 16 hexadecimal digit values are 0 through 9, as in the decimal system, and A through F for the remaining six values, 10 through 15 in the decimal system.

Example:

Convert several 4-bit binary numbers to their hexadecimal equivalents.

(i) 0011 binary = 3 hex
(ii) 1001 binary = 9 hex
(iii) 1101 binary = D hex

We already know that a binary number can be expanded by multiplying each bit value times the weight for its column position, and adding the results. Equivalently, each 4-bit group in the binary number can be assigned a single weight, the value of each group can be multiplied times its group weight, and the resulting values can be added together. The weights for each 4-bit group are of course expressible in powers of 2. However, they are also expressible in powers of 16.

Example:

Expand a 16-bit binary number to show the weights applied to each 4-bit group in powers of 2 and powers of 16.

$$0110\ 1111\ 1011\ 0011$$
$$= (0110 \times 2^{12}) + (1111 \times 2^{8}) + (1011 \times 2^{4}) + (0011 \times 2^{0})$$
$$= (0110 \times 16^{3}) + (1111 \times 16^{2}) + (1011 \times 16^{1}) + (0011 \times 16^{0})$$

Since 4-bit groups can be expressed as hexadecimal numbers having base 16 weights, we can immediately express any binary number as a number in the hexadecimal system.

Example:

Convert the binary number of the preceding example into hexadecimal and then decimal notation.

$$0110\ 1111\ 1011\ 0011 \text{ binary}$$
$$= 6FB3 \text{ hex}$$
$$= (6 \times 16^{3}) + (F \times 16^{2}) + (B \times 16^{1}) + (3 \times 16^{0})$$
$$= (6 \times 4096) + (15 \times 256) + (11 \times 16) + (3 \times 1)$$
$$= 28{,}595 \text{ decimal}$$

So a two-column hex number can represent an eight-column binary number, and a four-column hex number can represent a 16-column binary number. Clearly, hex notation is inherently easier to read than binary, once it is familiar. The largest number the Commodore 64 can handle in one operation is 1111 1111 1111 1111 binary, or 65,535 decimal, which in hex appears as FFFF. Table 1.2 showed all powers of 2 through this number plus 1. The power-of-16 table given as Table 1.6 shows all the powers of 16 through 65,536 decimal.

An abbreviation commonly used with numbers of different bases is to follow the number with the first initial of its number system. So a decimal number will be followed by a d, a hex number by an h, and a binary number by a b. If a suffix is omitted, the number is assumed to be decimal.

Example:

Show several decimal, hex, and binary numbers in their abbreviated form.

(i) 2017d = 2017 decimal
(ii) A7h = A7 hex
(iii) 101b = 101 binary
(iv) 101 = 101 decimal

Exercise:

Try these hexadecimal-code number conversions.

(x) 1010 1110b to hex
(y) BD7Fh to binary

TABLE 1.6 POWERS OF 16

Power of 16	Decimal value	Hexadecimal value
0	1	1
1	16	10
2	256	100
3	4096	1000
4	65,536	10000

Arithmetic. Because of the similarities between the hex and binary codes, we will discuss only hex addition and subtraction here. The other arithmetic and logical operations are similar extensions of their binary counterparts.

To perform totally hexadecimal addition you would have to memorize all the possible column additions. That is, you would have to immediately know that B hex + B hex = 18 hex, and so on. For most people it is easier to convert the hex digits mentally into their decimal equivalents for each column addition, and convert the column result back into hex. If the result is greater than 15d, you have to subtract 16d from it to obtain the carry for the next column.

Example:

Add the hex numbers A7B0h and 98A0h. Show all carry digits in parentheses.

```
  A7B0h
+ 98A0h
------
     0
   + 0
   ---
     0
     B
   + A
   ---
   (1)
     5
   + 7
   + 8
   ---
   (1)
     0
   + A
   + 9
   ---
   (1)
     4
```

Collapsed, the problem reads

```
  A7B0h
+ 98A0h
--------
 (1)
 405 0h
```

The leftmost carry in an addition result will often be discarded. If so, the numbers are said to *wrap around,* so that the first number higher than FFFFh is 0000h.

Hex subtraction is similar to the long method of binary subtraction that we discussed earlier. The two's complement of a hex number is found by subtracting the number from a zero value having a 1 "borrow digit" in the column left of its highest column.

Example:

Find the 2C negative of BC7Fh.

```
  1 0 0 0 0h
- B C 7 Fh
-----------
  F F F 10     (the carry is borrowed down to
- B C 7  F     the last column needing it)
-----------
  4 3 8 1h
```

or 4381h.

Exercise:

Solve the following subtraction problem by converting the subtrahend into a 2C value and adding it to the minuend. Don't forget to discard the carry from the result.

```
(z)  D B 8 A    minuend
   - 7 E C C    subtrahend
   ----------
```

BCD code. *Binary-coded decimal* or *BCD code* is a hybrid, as its name implies. Like the hexadecimal code, it divides a byte into two 4-bit groups. However, each group represents a decimal digit rather than a hexadecimal digit, and the weight of each group is a power of 10 rather than a power of 16. This implies that the values 0 through 9 are the only legal contents of a group, with the six values A through F a "no-man's land" that is never to be trespassed. BCD code, in short, is a binary way of expressing decimal numbers.

Example:

Show the BCD form of several decimal numbers.

(i) 27d = 0010 0111 bcd
(ii) 95d = 1001 0101 bcd
(iii) 34d = 0011 0100 bcd

In BCD code a byte's two groups of 4 bits can represent two decimal digits. Hence the largest legal value of a BCD byte is 99d. 100lb = 9d, so 99d appears as 1001 1001.

Imagine adding the two legal 4-bit nibbles 7 and 8. F, the sum, is clearly in the

no-man's land, so what do we do? There are six digits in no-man's land, so we add a 6 (0110b) to the result to "leapfrog" the forbidden territory. This addition generates a carry bit that must be added with the next-highest nibble addition, or that becomes the final leftmost digit if there are no further additions to perform.

The "no-man's land" correction is mentioned here only so that you will understand BCD arithmetic. You need not perform the addition or corresponding subtraction corrections, since your computer has special BCD operations that perform them automatically.

Example:

Add the two BCD numbers 0111 and 1000.

```
          0111 bcd     (7d)
        + 1000 bcd     (8d)
          ----
          1111       in no-man's land
        + 0110
          ----
(000)1    0101 bcd = 1 5 (15) decimal
```

BCD is used wherever the direct conversion of a number from its computer representation to a human-readable form is desired, or wherever each decimal digit must correspond to a single nibble, or most important, where precision to the right of the decimal point is required, as in accounting. Since many decimal fractions require an infinite number of binary fractional digits to express them, BCD is needed to produce an exact duplicate of the decimal number in machine form.

Exercise:

Add the BCD numbers 1001 1000 and 0101 0110.

(aa)
```
    1 0 0 1 1 0 0 0
  + 0 1 0 1 0 1 1 0
    ---------------
```

Some types of information must be coded *and* grouped. For instance, a force must be described with numbers for each coordinate direction in space. However, other types of information are of a simpler underlying structure and can be directly coded in imaginative ways. Although only the most obvious examples of these types of information have been coded, direct coding opens up some elegant and direct ways to handle all such information. Our last two categories of codes exploit this potential.

Symbolic Codes

Webster's defines a *symbol* as "an arbitrary or conventional sign . . . used in writing or printing relating to a particular field (as mathematics, physics, chemistry,

music, or phonetics) to represent operations, quantities, spatial position, valence, direction, elements, relations, qualities, sounds, or other ideas or qualities.''*

Symbolic codes are any codes that assign symbols to numbers. Anything that can be symbolized is potentially open for computer manipulation through coding. Commonly coded symbols include those for computer operations, letters, numbers, punctuations, graphic characters, and mathematical operations.

One can easily imagine codes for phonemes (i.e., the smallest distinguishing unit of speech), for the notes and instructions of traditional music, and for the particulars of other fields of human knowledge. The symbols in a symbolic code deserve careful organization according to their natural internal relationships to allow for the most natural mathematical transformations of their coded numbers. It would not make sense, for example, to design an alphabetic code assigning sequential letters to out-of-sequence numbers.

The basic computer principle of internal program storage, discussed earlier, originally relied on the development of a mill operations code. This was the first symbolic code developed, and the foundation of assembly language. It will be discussed at length in Chapter 3. The form of the operations code varies for different types of microprocessors.

The closest thing to a universally accepted symbolic code is a character code used in printers, terminals, telephone communications devices, and just about every large or small computer. Called the *American Standard Code for Information Interchange* or *ASCII,* it can be used in the Commodore 64 as well. To send an alphanumeric character to most non-Commodore printers or other devices, you send its ASCII number. Most devices you might attach to a computer have the built-in ability to interpret an ASCII number as the correct symbol. Commodore devices, however, use a variant of ASCII code, which is described below.

ASCII code uses only the lower 7 bits of a byte, with the d7 bit typically reset to 0. The d7 bit is sometimes used to indicate the number of 1's in the lower 7 bits. This added information is called *parity*. There are two types of parity, even and odd. Even, odd, or no parity is selected before data are sent. In *even parity* the d7 bit is given a value that will make the total number of 1-bits in the byte even. For instance, if there are three 1's in the lower 7 bits and the parity is even, bit d7 will be set to 1 to make the total number of 1-bits four, an even number. In odd parity the d7 bit is used to make the total number of 1-bits in the byte odd. So if there are three 1's in the lower 7 bits and the parity is odd, bit d7 will be reset to 0 to keep three 1-bits, an odd number.

Parity is used when sending ASCII characters to a device to check if any accidental bit changes (errors) have occurred in transmission. The receiving device is told the parity type before data are transmitted. If the receiving device receives a byte with the wrong number of 1-bits, indicating that one or more bits were changed during transmission, it asks the sending device to send the data again.

**Webster's Third New International Dictionary,* G. & C. Merriam Company, Springfield, Mass., 1971.

The 7 bits of the ASCII code allow 128 possible meaning assignments. In addition to numbers and letters, punctuation marks and control characters are coded. *Control characters* have noncharacter meanings such as "ring a bell," "this is the end of the message," and "back up a space." The most common control characters are the *carriage return* (0Dh), the *line feed* (0Ah), and the *tab* (09h). All ASCII control characters have number values between 0h and 1Fh.

The Commodore 64 extends ASCII for its internal use by defining the d7 bit of standard characters as the value 0 and by adding new characters with the eighth bit set to 1. It also substitutes its own graphic and control characters for the ASCII characters between 60 hex and 7F hex, and substitutes its own interpretations to the ASCII values 0h through 1Fh. We will call the Commodore ASCII code *extended ASCII* or simply *XASCII.*

These new assignments are not recognized by most devices in the outside world, so standard ASCII must still be used on occasion. A chart of the standard and extended ASCII codes makes up Appendix C. It omits those ASCII control codes that are of no use for normal programming purposes. Please glance through it and hold your place there for reference during the following remarks.

The control values unique to XASCII include CLR HOME, RVS OFF, and so on, as shown in Appendix C. These values duplicate the computer's keyboard functions and allow control of the TV screen directly from a program. Graphics characters can be sent by the program to the TV screen or to the Commodore dot matrix printer.

The Commodore 64 has several built-in "miniprograms," making up what is known as an *operating system,* to do frequently required chores. For instance, there is a miniprogram for sending single data bytes to locations of your choice, including the TV screen and the printer. If you provide this miniprogram with an XASCII value to send to the TV, that character, or its effect if it is a control value, will show up on the screen. Chapter 5 describes in detail the use of the operating system.

In Appendix C, note the relationship between the standard ASCII characters "a" and "A". Their values, 0110 0001 and 0100 0001, respectively, differ only in their d5 bits. This is true of all other standard ASCII upper- and lowercase letters. To convert from uppercase to lowercase, just add 20h to the uppercase value, or equivalently, set the d5 bit of the uppercase letter to 1.

There is a second Commodore code, called the *screen display code,* that represents most of the characters found in XASCII code. However, the XASCII control codes are omitted and extra graphics symbols are added.

These changes reflect the different usage of the screen display code. A program places screen display character codes directly into the storage locations from which the TV image data are obtained. With the Commodore 64's display of 25 lines of 40 characters each, a 1000-byte storage area holds the entire screen image. The screen display code is listed in Appendix B.

Actually, the screen display code is two codes collapsed into one. Depending on the current configuration of the computer, one or the other subcode will be used. So at a screen position holding the screen code value 1, either "A" or "a" can be

displayed, depending on the computer's configuration. Methods for using the Commodore codes in graphics are explained fully in Chapters 5 and 6.

Finally, realize that many types of symbols have never been coded. As character symbols can be coded, stored, and transformed inside the Commodore 64, so can your own symbols for art, pictographic languages, scientific symbology, and so on. Do not be reluctant to experiment with this underutilized code type.

Conceptual Codes

The last category of codes presses the limits of that which is directly codable. It has been largely ignored, but the act of describing it may challenge some to explore it.

Webster's defines a *concept* as "a general or abstract idea . . . a theoretical construct ('the *concept* of the atom')."* A *conceptual code* directly represents ideas instead of symbols or numbers. In general, the ideas are the elemental concepts of a field of knowledge. We can make a conceptual code of the whole spectrum of human emotion, for example, grading and ordering the varieties of joy, happiness, sorrow, and so on. We can even allow opposite emotions to be expressed as 2C or signed binary negatives of each other, as best serves their handling. A stripped-down version of this code might appear as follows:

Value	Meaning
0	Apathy
1	Like
-1	Dislike
2	Love
-2	Hate

Values in this code can be manipulated with simple arithmetic for interesting results. For instance, negation of a value also reverses the emotion it represents, incrementing a value yields a more positive emotion, and decrementing yields a more negative emotion. In a game program where the computer represents itself as a human being, such a code could be used to add emotional response to the computer's repertoire.

Other uses of symbolic codes are left to you, to discover new and efficient means for transforming abstract information seldom handled by computers.

Data Structures

All codes consist of single data elements such as numbers, characters, or elemental ideas. We all know of frameworks that hold data elements together. Your name is such a framework. It holds together a group of single letters. Such aggregates convey more information than is found in the sum of their individual data elements. A

**Webster's Third New International Dictionary,* G. & C. Merriam Company, Springfield, Mass., 1971.

name tells us much more than do its individual characters; a name identifies a human being. Similarly, an address composed of only letters and numbers helps us locate one building among all the buildings in the world.

A group of smaller data units forming a single and meaningful larger unit is called a *data structure.* Data structures are valuable because of the information beyond that found in their elements. The first type of added information is an overall interpretation assigned to the structure. Again, the interpretation "name" is assigned to collections of characters to identify people. This means of adding information should seem familiar since coding itself is the assignment of interpretation, albeit at a lower level.

The second type of information added to a data structure is the format given to the included data. The *format* implies relationships and the significance of individual data elements. For example, the data structure called a person's name is usually in a three-part format, with each part being a single-word data structure. By mixing the format order of the contained single-word data structures the meaning of the entire name structure is changed. The name "Mason," as a last name, indicates both the individual's kinship and his or her family roots in the profession of stone- or brickwork. The same name in the first position normally implies that the person is a male, a meaning absent in the last position. As a forename, "Mason" may also refer to strong qualities observed or hoped for in the baby, or more likely in our society, it may have been chosen for its sound.

So information can be expressed by assigning interpretations to numbers through coding, and by assigning interpretations to groups of data and to their internal organizations.

As the "name" structure suggests, data organization need not stop at one data structure level above the coded data elements. A data structure can be formed from other data structures. For instance, the common mailing-address data structure is made of the name data structure, the street-address data structure, the city, state-names data structure, and the zip code data structure. The interpretation of this structure as a mailing address is added information, as is the ordering of its member data structures. At an even higher level, a company-mailing-list data structure may consist of multiple mailing-address data structures, and so on.

Three levels of information representation are shown in Fig. 1.3. We view the

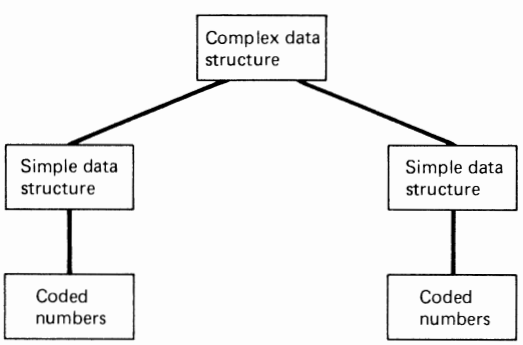


Figure 1.3

computer here as a mill that transforms numbers to perform tasks. The numbers represent information and are organized into data structures to represent additional levels of information. Arbitrarily complex information could be stored in data structures built of layers of lesser data structures. Thus we understand at an overview level what the computer does and what it does it on.

Just knowing what the computer does is insufficient. We must also know how it does it and why. To learn how to perform tasks with the computer, we must begin by examining the types of tasks that a computer can perform. We know that they must involve information. What are their other characteristics?

Types of Information Tasks

The two major types of tasks performable by computer are those that require the simple selection of action, sometimes called *switching tasks,* and those that require the transformation of information, called *data flow tasks.* Switching tasks are common in communications systems, where a microprocessor acts as a "traffic cop" and routes data from one place to another without transforming them in any way. Pure switching tasks are seldom encountered by the personal-computer programmer, so are not considered further in this text.

Data flow tasks are tasks that can be performed by transforming one or more known data elements or data structures into one or more new data elements or data structures of differing content or organization. Each input or output data element or data structure can be viewed as a separate flow into or out of the data-transformation process; hence the term "data flow task."

The computer's role in performing data flow tasks is similar to that of an electrical transformer in manipulating energy. An electrical transformer receives one or more electrical power flows of given voltages and transforms them into one or more electrical power flows of different voltages. In an analogous manner, the computer, under the control of a computer program, transforms input data flows into output data flows.

Data flow tasks lend themselves to pictorial representation. One type of data flow picture is called a *data flow diagram* or *DFD.* DFDs, however, are not just pictures; they are tools that help the programmer understand the problem and solve it. DFDs represent data flows as arrows and data transformers as circles. The generic DFD for a simple data flow task is shown in Fig. 1.4.

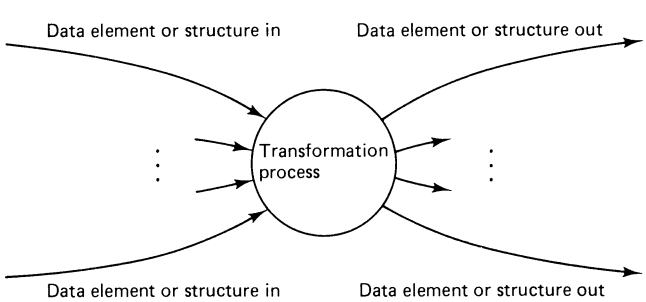


Figure 1.4

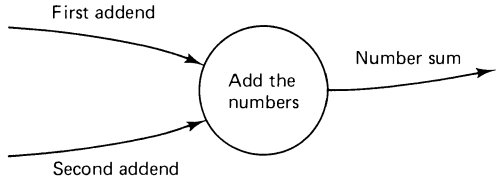


Figure 1.5

A simple example of a data flow task is the addition of two numbers. The input data flows are the two numbers being added, and the output data flow is the sum.

Example:

Draw the DFD for a two-number addition problem.

The diagram should look as shown in Fig. 1.5.

The "Add the numbers" process has no effect on the structure of the input data flows, since it produces an output flow of the same structure as the inputs. However, the contents of the output flow are (in general) changed from the contents of either input flow.

A transformation process that changes the structure of an input flow without changing the contents of its individual elements is also easily imaginable. The grain mill is an excellent physical example of this type of transformation. A grain mill takes raw grain and changes its natural structure by pulverizing it. All the grain's components remain, but they are reorganized. The same thing can be done with data structures by retaining their component parts while transforming the shape of the structure holding them.

Each transformation task can generally be broken into a group of less ambitious transformation subtasks. Each *subtask* transformation acts on portions of the input data flows or on data flows produced by other such transformations.

Example:

Show how the transformation process of a hypothetical data flow task might be subdivided.

Your diagram should look something like Fig. 1.6.

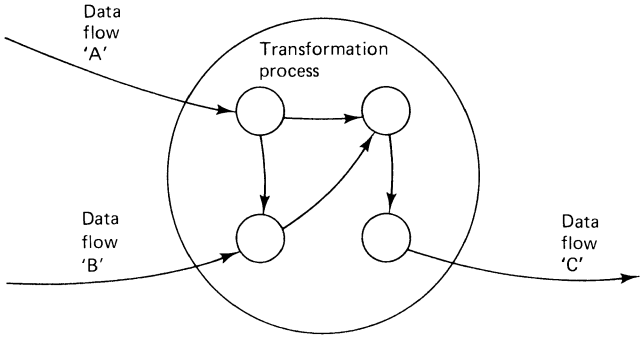


Figure 1.6

Thus a data flow task can be viewed as a single task or as a group of subtasks. The division of the one-task view into the multiple-subtask view, called *analysis,* is the key to designing a rigorous and thorough method to perform a data flow task. We will discuss data flow tasks and analysis again in Chapter 4.

We now know that the computer is the central component of the data flow transformer. We also know that a computer transforms data flows of one content and structure into data flows of different content or structure. What we do not yet know is why these things should be done.

What economy justifies the milling of data? Once again, the answer lies on a higher level of abstraction. Again we step up, landing this time on the concept of order.

Order and Entropy in Information

The concept of *order* is central to the functioning of the information mill. However, to define it we must refer to the closely related concept of *systems.* For instance, *Webster's* defines order in terms of systems, and implicitly defines systems as having order. We will invoke the definition of "system" to explain what order is, and then see why order justifies the use and existence of computers.

Webster's defines a *system* as "a complex unity formed of many often diverse parts subject to a common plan or serving a common purpose."* *Order* is the characteristic of a system that each of its parts is unique and separate from all its other parts, so that each part has a nonredundant role in the system. *Disorder* is the opposite characteristic: the degree of randomness or blurring of distinction between the parts within a system. The system "living organism" is ordered into distinct parts called "organs," "cells," "vessels," and so on. When disorder comes in the form of death, the parts of the body merge into homogeneous dust. A chunk of iron ore is part of the somewhat random and mixed system we call the earth, but when it is worked in a smelter, the ore is separated into its distinct and useful components: It is ordered.

The degree of disorder in a system is called its *entropy*. The greater the entropy, the less effective the system is at its purpose. Although there are scientific ways of measuring a system's entropy, we need not discuss them to consider how entropy affects information handling.

Entropy degrades information. For example, suppose that a national census is being taken street by street, with the result sheets stored as they come in. As new sheets are added, the pile becomes bigger, the records more jumbled, and the locating of any one fact increasingly difficult. The entropy of this system of information is increasing and the system is becoming less and less useful. If, however, the government hires an army of clerks to order the information alphabetically, the entropy of the information decreases and the system becomes available for more useful work.

**Webster's Third New International Dictionary,* G. & C. Merriam Company, Springfield, Mass., 1971.

Census information is not the only system whose entropy naturally increases with time. There is a law of nature called the "second law of thermodynamics," or simply the "law of entropy," which says that the entropy of any undisturbed system will always remain the same or increase with time.

The only way to combat the law of entropy is to disturb the system from the outside and reorder it. The army of clerks from headquarters served this purpose with the system of census information. The computer can serve this purpose with any system of information. It can either locate the causes of disorder and negate them, or it can discover the natural groupings of the system's parts and restore them. In either case it is able, locally, to reverse the law of nature and increase the usefulness of the information.

Suppose that you would like to send information over the telephone from your computer to another computer. The information will probably be ASCII encoded. However, phone lines tend to have stray electrical signals on them which can mix with the data being sent and confuse the receiving computer. Your computer sends an A, and the receiving computer reads it as a C, a difference of only 1 bit in the ASCII code. The message has become the victim of entropy. But the computers can negate this cause of disorder. The sending computer adds a data flow of parity information to the data flow of ASCII information. This additional information is placed on the d7 bit of the ASCII bytes, as explained earlier. The receiving computer checks the ASCII data against the parity data. If any part of the ASCII data has been changed, it is rejected and the sender is asked to repeat the ASCII data. This process continues until the entire ASCII flow has been correctly received. Thus the computer uses additional information to counteract the effects of entropy in a system and to transform a disordered ASCII data flow into an ordered and more useful one.

Computers can go one step further and increase the order of existing information. As we learned earlier, the organization of a data structure implies information about the nature and interrelationships of its data elements. A computer can often transform the structure of incoming data flows to produce outgoing higher-order information. This is a mechanization of the learning process. For instance, a computer can take the results of the population census and sort it by region. The resulting data structures contain only the original census reports, but their partitioning increases the order of the entire system. With this additional order it is easy to count the number of respondents in each region, to catalog their characteristics by region, and to obtain other information that was too difficult to find in the original overall sample. The computer has in a sense "learned" of new relationships among the data and "remembered" them as new data structures.

There are game programs that watch for a pattern in their human opponent's moves over several games. They treat this pattern information as a data flow and direct the computer to apply it to the data flow representing the best theoretical moves to create a high-probability guess of the human being's next moves. Clearly, this also is higher-order information.

Learning always decreases redundancy and the unprofitable repetition of

work. Looking through a stack of census forms every time you want a single fact is redundant, as is wasting an opportunity for attack by defending against a move that the opponent will not make.

All information tasks are performed by increasing the order of existing information, since information cannot be created from nothing. By consciously identifying the sources of entropy in an information system, and by considering methods for increasing its order, a task affecting that system can sometimes be performed more quickly or elegantly. At the least, knowing that entropy is at the heart of performing all useful tasks provides a deeper understanding of the program-planning process to be discussed in Chapters 3 and 4.

Thus entropy is the economy that justifies computers. At last we are ready to explore your electronic computer in depth. What does the blueprint for your information mill look like?

THE COMPUTER

To answer this question, we begin by drawing the information mill with its natural surroundings in the computer. Those surroundings are the grist storage area and the mill support systems for transportation and communications. Figure 1.7 illustrates the two major areas in the computer and how they are connected by the support systems. The transportation system carries grist back and forth between the mill and storage under the coordination of the communications system.

For the rest of this chapter we will explore the elements in Fig. 1.7. Do not be concerned about the specific means for controlling the computer; that is the realm of assembly language, discussed in Chapter 3. For now, your goal should be to get a feeling for what is inside the Commodore 64 and how those things work together.

The information mill portion of the computer subdivides into two parts. These two components correspond to the two operating functions of the mill: number transformation (i.e., performing the arithmetic and logical operations of the numeric codes discussion) and coordination with the mill world through a communications system. Reasonably enough, the number transformation section is called the *arithmetic-logic unit* or *ALU*. The coordinating section is named the *control unit* or *CU*. In this terminology, the information mill as a whole is sometimes called the *central processing unit* or *CPU*.

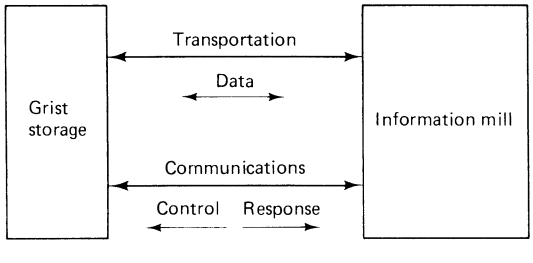


Figure 1.7

The CPU in the Commodore 64 is the 6510 *microprocessor.* "6510" is the manufacturer's part number for the device. The 6510 can obtain grist, or data, from storage locations. Since the only purpose of most of these locations is to remember data for the microprocessor's use, the entire data storage area has been dubbed the *memory map.* As we shall see, however, not all memory map locations serve as memory.

All the locations in the memory map hold 8-bit data, as does the microprocessor. Thus memory map locations are said to be one byte wide. Each memory map location is given an *address,* a number that identifies and sets it apart from all other locations. The address is used by the CPU to select the location as a data source or destination. With 8 bits as the standard data width, it would be convenient if addresses could be 8 bits wide. However, the 256 values allowed by a byte are insufficient to represent enough different locations for most purposes. The second-best width for an address is two bytes or 16 bits. Sixteen bits allows for 2^{16} or 64K addresses, as indicated in Table 1.2. The total number of addresses in the memory map is called its *depth,* so the memory map of the Commodore 64 is said to be 64K deep.

Both the memory map and the CPU represent data bits as voltages. Ideally, their circuits use two voltages, one at 0 volts or *ground,* and one at 5 volts or *power,* to represent the binary digits 0 and 1, respectively. Since some voltage shifting occurs in real circuits, these ideals have been loosened somewhat so that any voltage between 0 and 0.8 volt is interpreted as the binary digit 0, and any voltage from 2.0 to 5 volts is interpreted as the binary digit 1. The range from 0.8 to 2.0 volts has been reserved to separate the two legal ranges and has no digit interpretation. It is always avoided by the computer.

We now take a second cut at drawing the parts of a computer. Figure 1.8 shows one nonmemory function of the memory map: that of connecting the computer with the outside world. Such a connection is necessary to make the computer

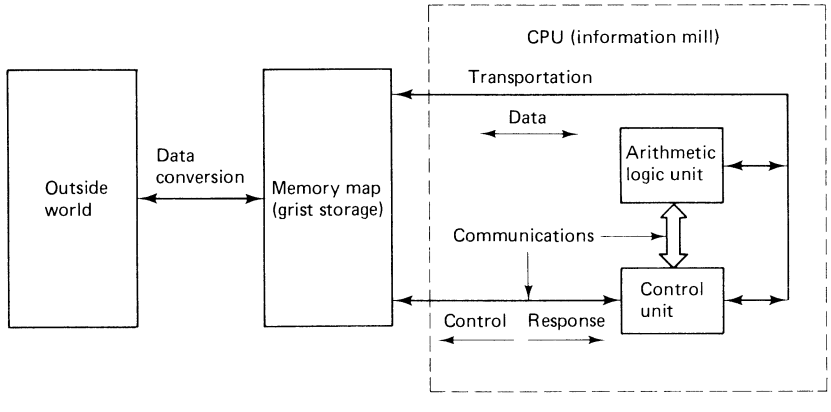


Figure 1.8

available for our use. Information received from the outside world must be converted into the digital form that the CPU can use. Similarly, information being sent from the CPU to the outside world must be converted into the form needed by its recipient. The central element in these or any other memory map accesses is still the CPU, so we begin our detailed look at the machinery of the Commodore 64 with this device.

The CPU

Some of the CPU's inner structure is already becoming apparent. The connections between the CPU and the memory map are the transportation and communications systems. Both systems originate in the CPU. Their information cargoes are of three types: addresses, data, and control signals.

Buses. The data transportation system carries addresses and data. Addresses are delivered to the memory map via a 16-wire route, and data are delivered in both directions on an eight-wire path. The communication system coordinates data transfers to and from the memory map. It sends coordinating commands and receives status responses or requests over a nine-wire path. These three groups of wires are called the CPU *buses*. The buses are the only connections the CPU has with the rest of the computer around it. They are called the address, data, and control buses.

Registers. A grain mill needs a few small local bins to hold the grain immediately before milling and to hold the milled product afterward for shipping. Similarly, the CPU contains a few small memory locations for short-term storage of addresses, data, and control information. These internal memory locations, called *registers,* are one byte wide and similar to those in the memory map. All registers connect to both a bus and a CPU section (the ALU or the CU). Both connections are needed since the registers are used for temporary storage when data are received or before data, addresses, or control signals are sent. Each register supports the function of its CPU unit.

Data registers connect to the ALU and the data bus. A data register receives a data byte from a memory map location during data input, and puts its contents on the data bus to send to the memory map on data output. A data register also *buffers* the data byte, which means that it holds the data byte in between transfers. There is even an intermediate buffer between the data bus and the data register, but it is invisible to the programmer and can be ignored for our purposes. The 6510 microprocessor has three data registers.

One of the 6510 data registers, called the *accumulator* and often labeled A, is the primary data register. It buffers the data byte coming in or going out, and does one more very important thing: The ALU, you may recall, takes one or two data bytes on input and transforms (e.g., adds) them to produce a one-byte result. The byte in the accumulator is usually one of those inputs. A memory map location sup-

plies the second input in most cases, and the result goes back into the accumulator. The process is shown in Fig. 1.9.

This is a hallmark of the 6510: Its organization revolves around an accumulator which provides an input to the ALU and receives the ALU's output. In other words, the 6510 is what is called an *accumulator-based processor.* One typical 6510 instruction says: "Logically AND the contents of the accumulator with the contents of memory map address z, and deposit the results in the accumulator" (z is specified in actual use, of course). In Fig. 1.9, the ALU internals are represented by arrowheads to illustrate the two data inputs and one data output.

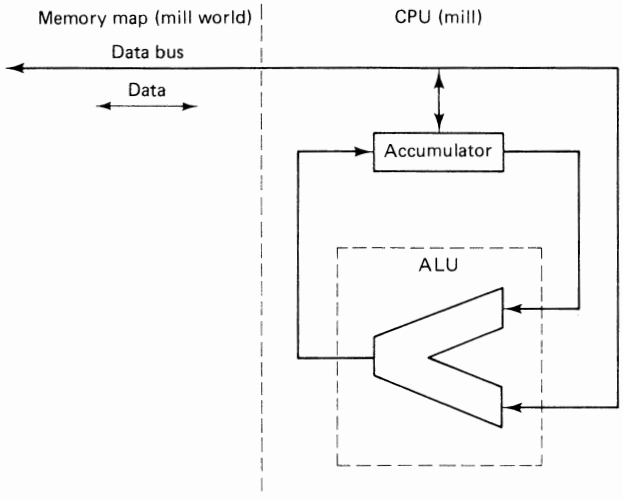


Figure 1.9

The remaining two 6510 data registers are called *index registers* and are labeled X and Y. They too can buffer data, and in a more limited way, serve as ALU inputs. Their main use, however, is related to addressing, a topic that will be discussed later. The three data registers all connect with the ALU (Fig. 1.10), so they are traditionally considered as part of the ALU section of the microprocessor.

There is a special-purpose register that is also closely associated with the ALU. Recall that under program control, the CPU can look at the results of previous operations and make program decisions based on them. These results represent the status of the CPU due to its recent history. A special-purpose register maintains records of these results and is therefore called the *processor status register,* labeled P.

The processor status register keeps a record of important arithmetic outcomes such as zero or negative results from subtractions and overflows from 2C additions by setting *flags* or register bits. The flags can be tested and reacted to by the CPU under program control. The processor status register is more commonly called the *flag register* because it flags specific types of CPU operation results with set bits.

Other flags in the flag register show if the CPU is currently set up for BCD or binary arithmetic, if there has been a carry from an arithmetic operation, and if a

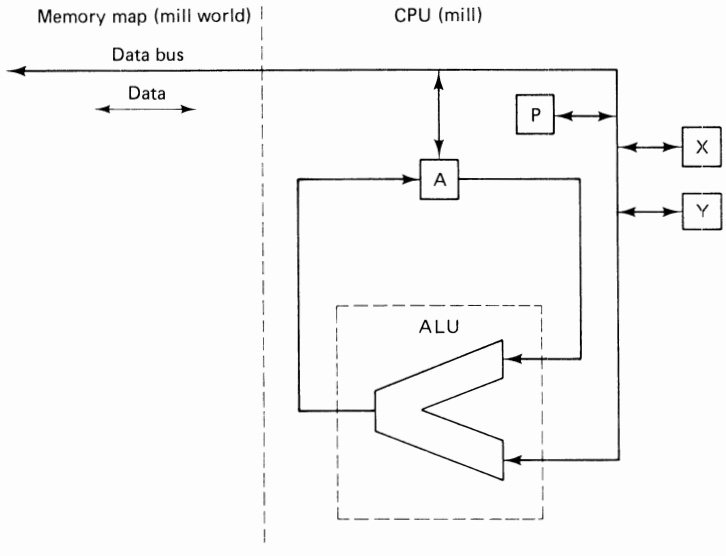


Figure 1.10

break or interrupt, two means that we have not yet encountered for changing the flow of program execution, has occurred.

The first four registers that we have discussed—the accumulator, the X and Y registers, and the status register—all connect to the data bus and the ALU. The next three registers connect to the address bus and the ALU, and hence are called *address registers.*

Address registers place their contents on the address bus, but never accept data from it. Two of them can be grouped together to hold the 16-bit addresses used to select memory map locations. The third register is used by itself in a special way that will be explained shortly.

If you recall, one of the greatest breakthroughs in the development of computers came from placing the program into memory. This was made possible by the coding of program operations, so the operations and their operands could then be loaded into memory. To execute the program, the CPU need only point to the location of the next byte of the program and fetch its contents to the mill. If the byte represents an operand it will be milled, and if it represents an operation it will be interpreted and executed.

This pointing to a memory map location is what we call *addressing.* The address located in the address registers is placed on the address bus and sent to memory. Fractions of a millionth of a second later, a "write" or "read" command is sent over the control bus to select and activate the addressed memory location for data transfer in or out over the data bus.

The two general-purpose address registers are grouped as a single 16-bit register called the *program counter* or *PC.* The upper 8 bits of an address are held in

the PC High or PCH register. The lower 8-bit register is called PC Low, or PCL. The PC is an *incrementing register;* that is, it is a register that automatically adds 1 to its own contents under certain circumstances. To simplify our explanation it is also preferable to think of the X and Y registers as being able to both increment and decrement their own contents. Increments and decrements are the only mathematical operations that can be performed directly on the contents of the X and Y registers.

The PC addresses each byte in a program as the program executes. After each program byte is fetched, the PC is incremented or increased by 1. In this way the PC counts its way up through the memory locations holding the program.

Before the contents of the PC exit the CPU, they are buffered by another two "invisible" registers. As with the previous invisible register, these cannot be directly affected by the programmer. We mention them because they allow addresses from sources other than the PC to be placed on the address bus. These two registers are labeled HIBUF and LOBUF in Fig. 1.11.

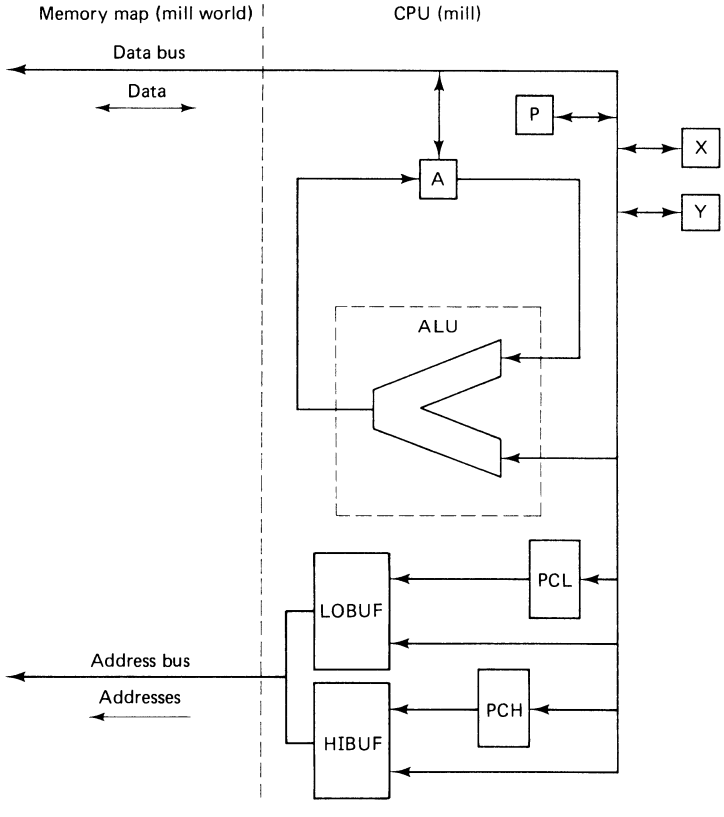


Figure 1.11

These other sources will be discussed as we go along, but one of them can be mentioned now. It is possible to retrieve a 16-bit address value from memory and load it directly into the invisible registers through the data bus. This allows a program to access data locations that are totally removed from the memory area holding the program, and thereby to store data anywhere in memory.

How is the PC loaded when the computer is first turned on? As Figure 1.11 suggests, it is loaded from the data bus. When power is first applied, a CPU *startup sequence* loads PCL with the byte contents of memory address FFFCh and loads PCH with the byte at FFFDh. These bytes form a 16-bit address for the mill to start from. The CPU assumes that there is a program byte at the new address in the PC, and begins execution from there. You will not be involved in this process since Commodore has placed the necessary address data and program operations in permanent memory.

The PC is normally incremented during program execution to access the sequence of bytes in the program. There is a way to load the PC with a completely different execution address, however. If during the execution of a program, an instruction called a JUMP is encountered, the two operand bytes following the JUMP byte in memory are loaded from the data bus into the PC. The new PC contents then address memory as usual, and the program continues execution starting at the new address.

An additional twist is the *conditional jump* or *branch*. This instruction says "jump *if* condition M has occurred." The condition could be a zero result from a preceding subtraction or many of the other conditions recorded in the flag register. The specific conditional jumps are discussed in Chapter 3. Using branches it is possible to interrupt the straightforward incrementing of the PC and point it to the starting address of some other program path.

Programs address data by loading HIBUF and LOBUF directly, as we have already discussed. A fast variation on this scheme is to retrieve just one byte from memory to load LOBUF and to set HIBUF to all zeros. Although this addressing mode limits the number of accessible addresses to the first 256 or 100 hex in the memory map, its speed allows us to think of these locations as internal CPU registers that greatly expand the 6510's capabilities. Each 100h addresses in memory are known as a *page* of memory, numbered starting with *page zero* from 00h to FFh. Thus this type of addressing is called *page-zero addressing*.

The third address register remains to be explained. It is a solitary register that is loaded into LOBUF. HIBUF is loaded with the value 0000 0001b or 01h, which allows addressing from 100h to 1FFh. In page terminology this register allows addressing of page 1. However, the way this register is used in programs is what distinguishes it. We illustrate its function with an analogy.

A careless grocer notices that the milk rack in the cooler is almost empty. He calls the milk delivery company, and by and by a truck pulls up. Ten gallons of milk are delivered, and the grocer, too lazy to take it into the back room, inserts the milk

on the rack from the front (pushing back the old milk). You come shopping and want some milk. You pick up the milk carton at the front of the rack, and since it was placed there most recently, you get nice fresh milk. This grocer always fills the rack from the front, so the milk at the back of the rack stays so long that it curdles.

Now, no grocer is going to be so careless as to put the fresh milk up front, but this fantasy demonstrates a principle. The latest milk carton placed on the rack by the grocer is always the first milk to be removed. Think of the rack of milk as a horizontal "stack" of milk. We can say that "the last milk in the stack is always the first milk out."

Why not do the same thing with data? Suppose that in the course of some calculation in a program, you need to store several numbers for later use. If you set up a "data stack," like the milk stack, the numbers can simply be "pushed in at the front" as they are generated, and "pulled off the front" in reverse order as needed.

One of the advantages of this type of data transfer is that only one address is needed: the location of the top, or latest front position, of the stack. In practice, the address of the top of the stack is kept in the third address register. Because it services the stack, it is called the *stack pointer register* or simply the *stack pointer.*

To build a stack, a starting or bottom address must first be chosen and loaded into the stack pointer. Since no data are in the stack, this address is also the top of the stack. The first data byte is then placed into the accumulator. Finally, execution of a 6510 instruction called a PUSH causes the accumulator contents to be placed into the location whose address is in the stack pointer, and the address in the stack pointer to be decremented by 1. Subsequent pushes force the stack pointer addresses even lower, causing the stack to grow downward. This seems a little unusual to most people, since it appears more natural that the stack would grow toward higher addresses. However, the stacks for most microprocessors grow the same way.

To retrieve the last byte pushed on the stack, the stack pointer must first be incremented by 1 to make up for the decrement after the last push. Then the contents of the location pointed to by the stack pointer can be loaded into the accumulator. The 6510 instruction causing this operation is called a POP.

Since the stack pointer can address only locations 100h to 1FFh, the stack can at most hold only 256 bytes. This, however, is usually more than enough. 1FFh is frequently chosen as the bottom of the stack to allow for this maximum size, but it is occasionally useful to set up more than one nonoverlapping stack, each with its own bottom address in page-one memory. In that case, one must load the current top address of each stack into the stack pointer before use, and save it before switching stacks. The assembly language instructions needed for these two operations are discussed in Chapter 3.

One use of the stack is to save the contents of the data registers prior to execution of a part of a program called a *module* or *subroutine.* Modules are also discussed in Chapter 3. The accumulator and flag registers are the only locations whose contents can be placed on the stack or retrieved from it. The other registers and all the memory locations must go through the accumulator to use the stack.

The stack register can be incremented or decremented by POP and PUSH instructions, and the PC register can be altered by addition of 2C constants. HIBUF and LOBUF can be altered by the addition of values from the X and Y registers. The different ways of altering all these address registers provide the different addressing methods or *addressing modes* that make the 6510 an extremely flexible microprocessor. Addressing modes will be discussed again in Chapter 3. Since the contents of the address registers can be altered with ALU operations, the address registers, like the data registers, are associated with the ALU section of the CPU (Fig. 1.12).

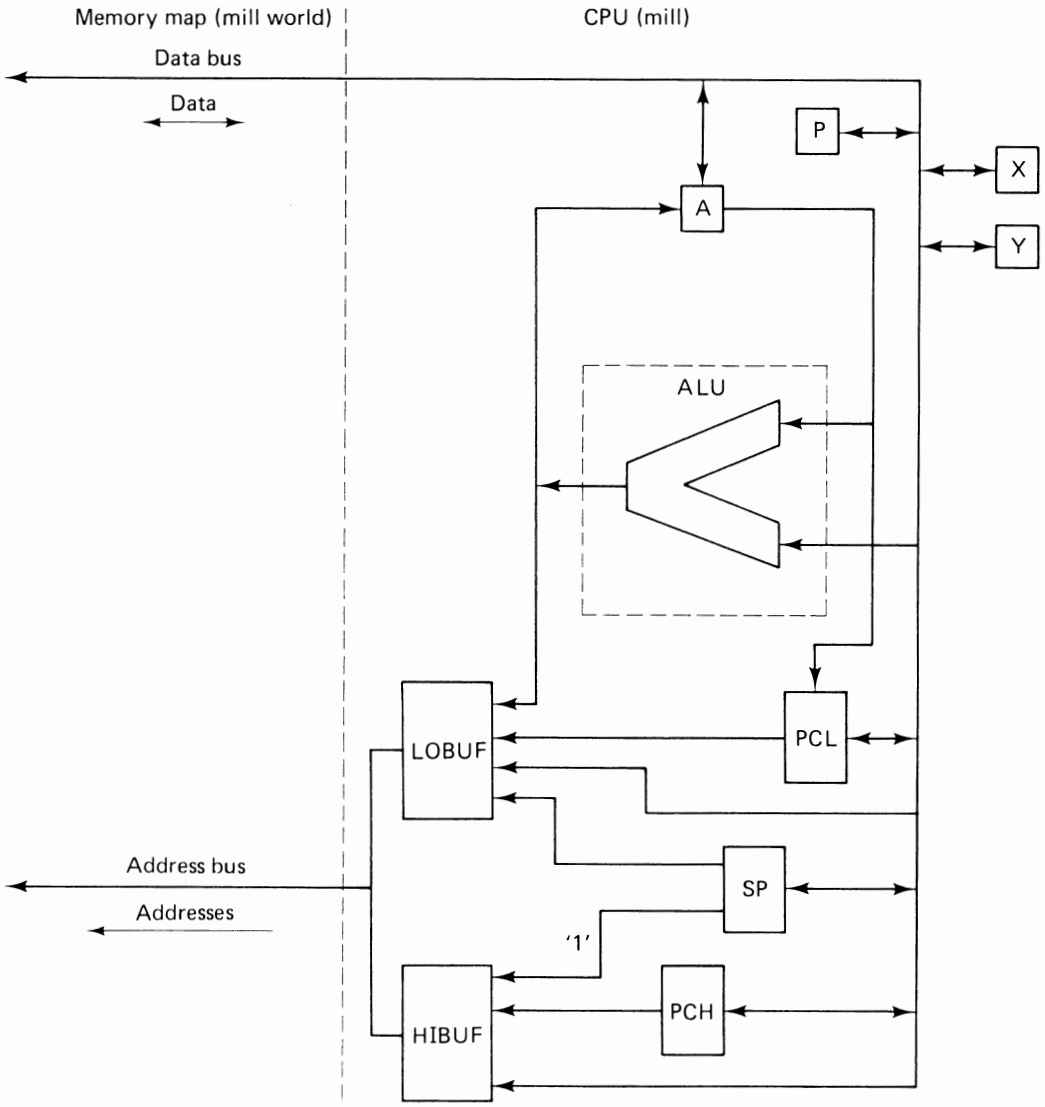

Memory map (mill world)
CPU (mill)
Data bus
Data
P
X
A
Y
ALU
LOBUF
PCL
Address bus
Addresses
SP
'1'
HIBUF
PCH

Figure 1.12

Yet another type of register is physically located in the 6510 but is accessed by the 6510 as if it were in the memory map. Two registers fit this category. The bits of one register are connected with *pins* or wires exiting the 6510 microprocessor. The other register determines the data direction of each bit in the first, that is, whether each bit will be an input to the 6510 or an output. Thus one register is an *input/output* or *I/O* register, and one is a *data direction register* or *DDR*.

These registers are used to change the types of locations in the memory map. As we stated earlier, memory is not the only type of location in the memory map. The Commodore 64 contains 64K of user memory alone, with many more locations of other types available. Since the 6510 can address only 64K of memory map locations at a time, using additional locations requires the ability to attach and detach them from the CPU buses. Three bits of the I/O register perform this task. We could think of them as a *memory-map configuration selector.*

Another 3 bits of the I/O register control and monitor the 64's optional data storage cassette player. The last two bits are unassigned. This register is the *configuration register,* labeled CR in Fig. 1.13. Since this register pair can move data in or out of the CPU, it is jointly called the *bidirectional I/O port.* You can think of the bidirectional I/O port as being associated with the ALU since it can be loaded from and stored into the accumulator, but the relationship is more tenuous than it is for any of the other registers.

The last type of register is connected directly to the CU and the data bus and indirectly to the control bus. There is only one register of this type and it is called the *instruction register,* labeled IR in Fig. 1.13.

A program is composed of instructions built of one operation byte and zero to two operand (address or constant data) bytes each. Every time an operation byte is brought into the CPU, the data bus deposits it in the instruction register.

The instruction register is attached directly to a control unit (CU) circuit called the *decoder.* The decoder converts the operation byte into explicit directions that select the CPU registers, timing, and actions needed to execute the instruction. The control bus then delivers any directions affecting the mill world or memory map around the CPU.

Each operation byte specifically informs the CPU of the number of following operand bytes, so the CPU knows to look for the next operation byte after the last operand in the present instruction. With the additional constraint that a program must always start with an operation byte, the CPU always knows if it has retrieved an operation or an operand. The mystery of how the CPU understands an instruction is solved.

This completes our discussion of the floor plan, or structure, of the CPU in your computer. Now let's fully expand the CPU side of the mill world first seen in Fig. 1.7.

The fetch-execute cycle. You have gained a pretty sophisticated idea of how the CPU is put together. Now we can discuss how it works.

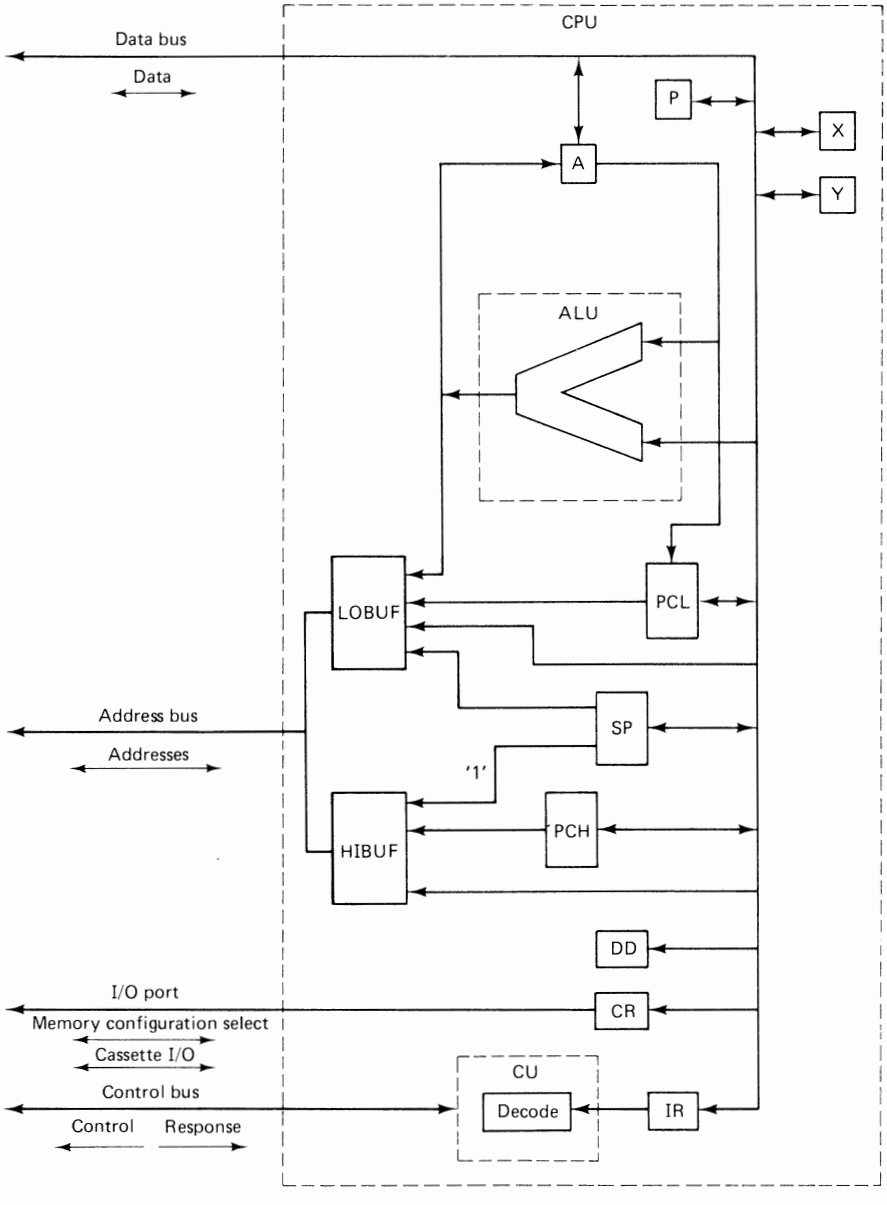

CPU
Data bus
Data
P
X
A
Y
ALU
LOBUF
PCL
Address bus
Addresses
SP
'1'
HIBUF
PCH
DD
I/O port
Memory configuration select
CR
Cassette I/O
CU
Control bus
Decode
IR
Control
Response

Figure 1.13

The CPU acts out a cycle that repeats endlessly from the time the computer is turned on until it is turned off. In a way this cycle resembles the cycle of life; inexorable, unstoppable, it defines the CPU the way that birth and death define things animate. It is called the *fetch-execute cycle.*

As its name implies, this cycle can be divided into two phases, called fetch and execute. The names are evocative. In the *fetch* phase of the cycle the operation byte of an instruction is retrieved from program memory and decoded. In the *execute* phase of the cycle the operations ordered by the operation byte are carried out and affect any operands accompanying the operation. We will walk through a fetch-execute cycle, starting with the fetch phase.

At the start of the cycle, the PC holds the address of the next program instruction. This address is loaded into the address buffers LOBUF and HIBUF and placed on the address bus. A "read memory" signal is sent on the control bus. Less than half a millionth of a second later, the contents of the storage location at that address are on the data bus. The CPU, knowing that it is retrieving an operation byte, loads the contents of the data bus into the CU's instruction register. This completes the fetch phase.

There is a very short delay while the decoder circuit of the CU breaks the instruction byte into its constituent parts. As we mentioned before, these parts select the registers used by the instruction and the actions to be performed. Then the execute phase begins.

From zero to two additional locations in memory are read to load the data needed for the specified operation. Zero reads are needed if the operation requires just one byte from the accumulator, or if it requires no data at all. One read is needed if the operation works with two data bytes: one from the accumulator and one from the memory location following the instruction byte. Finally, two reads are needed if the operation works with two data bytes: one from the accumulator and one in a memory location whose base address is in the byte or two bytes following the operation. In the latter case, the base address from the program is prepared by the CPU in ways that we shall study shortly, placed in LOBUF and HIBUF, sent out over the address bus, and the data value is read.

Once any operands are in place in registers or on the ALU memory input, they are operated on according to the directions of the operation byte. When this task is complete, any ALU output is sent to the accumulator or to LOBUF, with the flag register usually being affected. The execute phase is then complete.

The CPU can send data to memory as well as retrieve it. This is called a CPU *write.* An address is placed on the address bus, a "write to memory" signal is sent on the control bus, and the CPU places a data byte on the data bus. Again, the memory takes just under half a millionth of a second to respond, and then a timed control signal from the CPU tells the addressed memory location to receive the data. This capability allows the CPU to alter memory locations and store new data generated during the running of a program.

The sequencing of execution steps is synchronized with a *clock*. It "ticks" approximately 1 million times each second, and every instruction consumes a set number of ticks, called *clock cycles*. Clock cycle information on each instruction is included in Chapter 3. A program or a section of a program that must execute very quickly will reward the programmer's attention to choosing minimum clock cycle instructions and instruction sequences.

The Memory Map

As we have said repeatedly, memory is only one part of the memory map. There is a second major type of memory map location, called I/O.

Memory locations come in two varieties. One type allows the CPU to both read from and write to it, while the other allows the CPU only to read its contents. The first variety of memory is abundant in the Commodore 64. The "64" in "Commodore 64" refers to the 64K of this type of memory in the computer. The most logical name for this memory would be "read-write memory," but the historical name has been *random access memory* or *RAM*. This name arose when certain types of memory could only be read or written in sequential order, so memory with locations accessible in any order (randomly) was dubbed RAM. RAM must have power applied to retain its contents. When the computer's power is turned off, any data in RAM are lost.

The second variety is more descriptively named *read-only memory* or *ROM*. Although the CPU cannot write data to ROM, it can read ROM randomly in the same way that it reads RAM. Thus both RAM and ROM are random-access memories, making the name RAM a particularly poor choice. The CPU addresses RAM and ROM the same way, which leaves it up to the programmer to make sure that a program's use of the two types of locations makes sense. The contents of ROM are retained whether or not power is applied. Any programs or subprograms needed to run the computer have been placed in 20K of ROM in the Commodore 64. Included in the ROM are the BASIC programming language and the operating system.

The Fig. 1.8 mill-world drawing shows the outside world connected to the memory map. This connection is made through the second major type of memory map location, the *I/O port*.

There are read-only, write-only, and read-write I/O locations. All I/O ports connect to CPU buses and to devices or circuits outside the CPU and the memory map. They buffer data from the CPU and pass it on to an outside device, or in reverse, they buffer the data from the outside device and hold it for the CPU to read. These locations are called "ports" because data enter and exit the computer through them. The internal 6510 I/O register is a port of sorts, since it connects the CPU with other nonmemory circuitry.

Data are transferred directly through I/O ports to and from large-capacity storage devices such as disk drives and cassette recorders. Other peripherals, such as printers, game joysticks and paddles, light pens, and talking data-to-voice converters, communicate through the I/O ports. The modem, a device that connects computers by telephone, attaches to the computer through an I/O port. Commodore markets most of these peripherals, and we discuss them at greater length in Chapter 5.

Although the just-mentioned peripherals all accept bit-pattern or digital data, there are other peripherals, such as TV and audio equipment, that require signals in "wave" or analog form. The circuitry that provides these analog signals is connected to and packaged with several of the I/O ports.

Almost all of the electronic circuits in the Commodore 64 come grouped on *integrated circuits,* called *chips* for the pieces of silicon on which they are placed. There are eight 1-bit-wide 64K-deep RAM chips in the 64, which together form the 8-bit-wide 64K-deep RAM memory, and three 8-bit-wide ROM chips: One 8K-deep ROM holds the BASIC programming language, another 8K-deep ROM holds the operating system, and the third 4K-deep ROM holds codes for the visual appearance of characters. The computer uses the latter code values to display characters on the TV.

Many of the chips containing I/O ports also contain circuits to convert between the port data and the signals needed by outside devices like the TV. The design of this part of the 64 is simple and elegant. There are only four chips, two identical, controlling all of the aforementioned peripherals. The chips handling video and audio have on-board I/O ports and analog outputs; data written into these ports control the analog output of the chips. Thus digital data produced by a program control both analog and digital devices through the I/O ports. The graphics and audio chips and the supporting principles of the visual and aural arts are discussed in Chapters 6 and 7.

Summarizing our discussion of the physical contents of your computer, we note that it contains three major types of chips: memory, I/O, and the microprocessor.

We can now expand the left or memory map side of the mill world diagram in Fig. 1.8. The "conversion" process in Fig. 1.8 is handled in Fig. 1.14 by the I/O chips on which the I/O ports reside. All three buses from the CPU affect all devices in the memory map.

The great founding computer principles of number codes, internal program storage, and a central information mill with a surrounding world of storage, transportation, and communications are now in your hands. Domination is next. Programmers aren't built in a day.

Exercise:

Answer the following questions to review the internal structure of the C64.

(bb) What are the three CPU buses?
(cc) What are the two sections of the CPU?
(dd) What are the three sections of the memory map?
(ee) Name the CPU data registers.

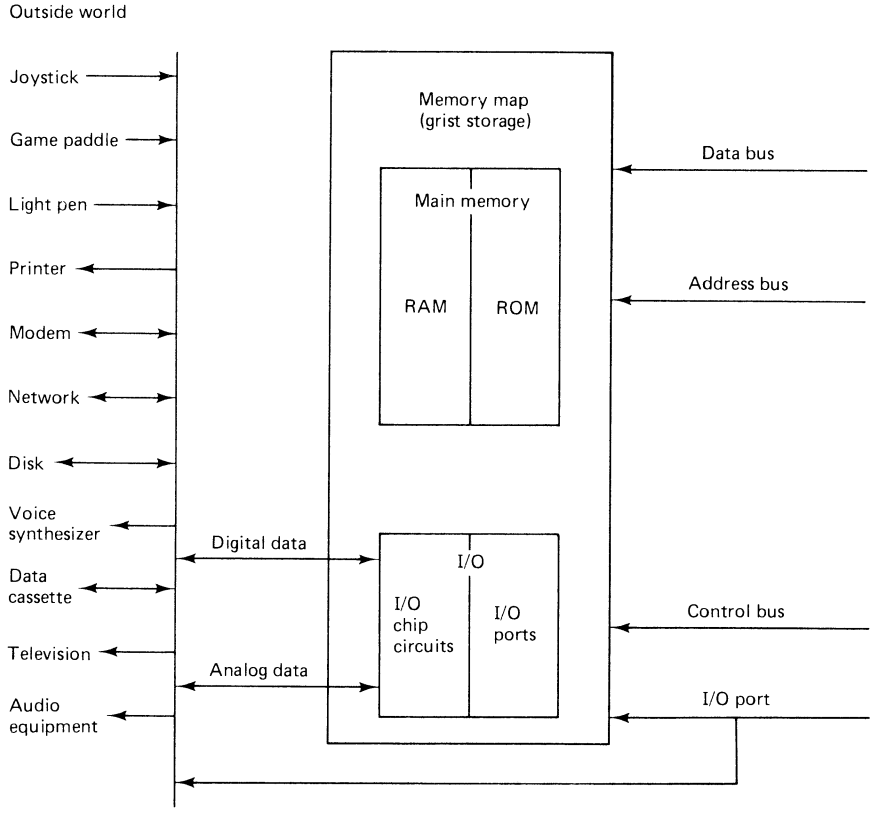


Figure 1.14

FOR FURTHER STUDY

Subject	Book	Publisher
C64 circuitry	*The Commodore* 64 *Programmer's Reference Guide* (schematic & chip spec appendices)	Howard W. Sams & Co., Inc. and Commodore Business Machines Inc.
Computer architecture:	"System Architecture" by John Zarrella	Microcomputer Applications

CONCEPTUAL QUICKSILVER: DATA STRUCTURES

hg

clear fractal islands spin away

glacier erupting silvermilk

fjord-tide over

shards

now mirrors

glint of hard air

stranding icebergs on bottomland

Mercury, or "quicksilver," is the densest element that is both liquid and stable. Disturb a little of it and you move a lot of mass. The designer of a mass pump might well choose mercury as its mass-transporting fluid.

Now imagine a similar fluid that carries dense "conceptual mass." It dissolves information and then compactly transports it. In short, it is a "conceptual quicksilver." The quicksilver is the data flow, and it consists of information "molecules" called *data structures*. As was explained in Chapter 1, data structures are highly organized groupings of coded information.

In this analogy the CPU is a transmutation device that includes a pump to cir-

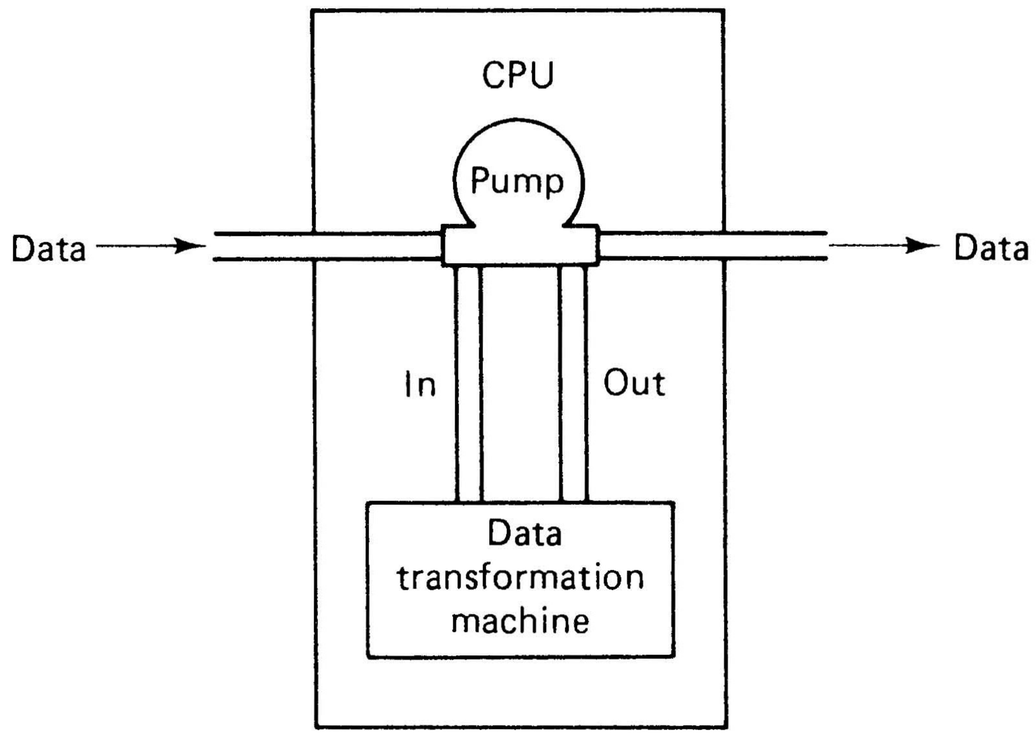


Figure 2.1

culate the fluid before and after its molecules are transformed (Fig. 2.1). The transmutation process consists of number transformations controlled by a program. Most microprocessors are limited because they can only directly address, and therefore easily move and alter, a few types of data structures. The Commodore 64's microprocessor is more flexible. It directly addresses most types of data structures, so it can perform more types of data transformations more easily. This places fewer limitations on the programmer and simplifies programming.

In Chapter 1 we explored what computers are, what they do, and why they exist. After answering those foundational questions, the next most important thing to understand is the material that your machine is designed to handle. That material, data structures, motivates every aspect of computer operation and use. From Chapter 3 on, everything we will discuss is influenced by and best understood in terms of data structures. The more thoroughly you understand this subject, the more prepared you will be to master the complete method for "power programming" the Commodore 64.

MODELING

Data structures are derived from raw information using a technique called *modeling*. Modeling is a two-stage process that can be applied to any system. It begins with analyzing the system by breaking it down into its most basic pieces. It finishes with synthesizing or combining the parts to form a representation of the most efficient version of that system for a particular purpose. If the system being modeled is a body of information, these two stages are called *data selection* and *data structuring*, respectively. These stages will be discussed next. If the system being modeled is a task that transforms the data, the two modeling stages are called *analysis* and *design*. They will be discussed in Chapter 4.

Data Selection

Identifying the information that is applicable to a given task can be a difficult job One approach is to identify a category that encompasses everything you want to know and no more. For example, consider the male member of a softball team who

has never yet played in a softball game; he always sits on the bench. Can we categorize him as a softball player? On the other hand, can we afford not to, given that he may play tomorrow?

The team member's role is actually well defined. Our difficulty comes from placing him in a category that is too simple to encompass all the relevant information about him; primarily, it omits the fact that he's a reserve player.

Problems with identifying the information needed to perform a task often result from such attempts at oversimplification. Problems can also result from "overcomplexifying" including so much specialized information that the resulting category applies to only one ridiculously restricted situation. For instance, we could place the softball-team member in the category "overweight but enthusiastic used-auto dealer and sometimes Joe DiMaggio fan who has never played ball on the field but who has played catch five times and swung a bat twice, whose name is Fred Bazooka, whose wife's name is Dotty, who has three kids named Joe, Annie, and Marvin, and a sister named Griselda, which means 'gray battle-maiden' in old German, and the name really fits" (Fig. 2.2).

The latest category includes much more information about its subject than the ballplayer category. However, a lot of it is inapplicable to the question we are really interested in: Fred's status on the team. Further, no one but Fred will ever fit in this category, so we cannot collect parallel information about the other players on the team and come to any conclusions about Fred's role on the team.

Figure 2.2 Fred and his family.

The type of category that is most useful is one that is simple and broad enough to be applied to other similar objects (e.g., team members) but detailed and complex enough to contain all information needed to answer the questions we are interested in. If that category is then made into a data structure and stored on a computer, a program can ask those questions for us about many different subjects. An appropriate category for our softball-team member might be "team-member status." This category could contain the following information on any team member: name (Fred), whether the member is reserve or active (Fred is reserve), how many games the member has played this year (Fred has played 0), how many games the member has ever played (again, 0 for Fred), playing position, batting average, home runs in each game, and so on. A program handling the team-member status data could compile team and even league statistics, as well as tell us where Fred stands relative to the other team members.

The first step in performing an information-handling task, then, is to identify the information that is required to produce the answers we want. This is the crux of the first stage of modeling: selecting the smallest amount of information from which the desired information results can be derived. This narrowing down is also sometimes called *abstraction.*

How do you select the right subset of data to gain desired results? Sometimes the choices are obvious, but sometimes you must just make a decision and proceed. Fortunately, a poor choice can be abandoned and corrections such as adding or deleting data made later. All programmers backtrack on occasion, and often more than once on a given task. Overcoming any reluctance to improve a model is one of the most important steps you can take toward fine craftsmanship in programming.

Data Structuring

Once you have a minimal set of data, you must decide on a structure in which the data fit most naturally. We have a similar problem with structure that we had with raw information. From the complex structure organizing all imaginable information about a subject, we must identify the minimal structure necessary for performing the desired information-handling task. A good criterion for this choice is *efficient processing.* Whichever structure makes the processing of that data by a program most efficient, whether the standard of efficiency is program size, speed, or whatever, should be used. This is a question you will be better equipped to answer once we have discussed assembly language and program planning.

Example:

Walk through an everyday example of the modeling process.

The models most of us are most familiar with are the airplanes, boats, and so on, we grew up with. The two stages of modeling are used by their designers.

First, designers select a subset of the total information about their real-world subject. Most model airplanes include a canopy, but few include a rotating shaft inside the jet engine. It is economical for designers to include the minimum amount of information necessary to convey "jet airplane." This is a kind of minimal-data choice.

Second, a structure, an organization of parts and method of assembly, is chosen for the model. This structure probably has little to do with the real-world object's structure, but then, it doesn't need to; the kit's structure should be best suited to the purposes of the kit. This means that the model designers strongly consider simplest processing (i.e., simplest manufacture and kit assembly) in choosing the model's structure.

Finally, by using the manufacturer's minimal set of parts and implementing the suggested structure, you obtain the desired result: a reasonable copy of a real airplane. That is, unless you are like many of us who have fouled a kit's structure with globs of glue and misplaced parts: Who cares if you're going to blow it up with firecrackers anyway? Even this type of mishap is instructive: programs can be just as messy when *their* data structures are neglected. The difference is that fouled programs often blow up on their own.

DATA STRUCTURE TYPES

Three basic data structures are all that are needed to organize any data for processing. Called the *sequence,* the *repetition,* and the *selection,* they have the simplest structural forms known.

Data structures and individual data items can be defined in a *data dictionary. Data definitions* use a simple notation based on data names and symbols for the three ways of organizing them. This notation resembles equations, except that the name of the data structure being defined is placed on the left of the equals sign, and the names of the data elements (data structures or data items) contained in the structure, and the symbols for the structure organizing them, are placed on the right. The symbol for each type of data structure is discussed in that structure's section below.

Sequences

In a sequence structure, component data elements of differing types are placed in a simple succession, much like boxcars in a train. There are no other type or structure relationships between the data elements, making this the simplest of all data structures.

The mailing-address data structure of Chapter 1 is of this primitive type. Certainly, an alphabetic name and a numeric zip code have no similarity or relationship, except through their colocation in the sequence "mailing address." Since both elements are of different types, computer processing of the mailing address will probably require different actions for each element.

In general, every item in a sequence will have to be processed in an independent way. So the part of the program that processes a sequential data structure will itself consist of a sequence of different actions, executed one after the other. Processing structures almost always match the data structures being processed.

The symbol showing the sequential structure relationship between data elements is the plus sign. Thus the definition for a sequential data structure consisting of three data elements would be of the form

```
data_structure_name = element_1_name + element_2_name +
                      element_3_name
```

In an actual definition each label would be replaced with the descriptive name of the data.

An individual sequence structure is called a *record*. Each data element in the record is called a *field*.

The sequential data structure can be represented pictorially as shown in Fig. 2.3.

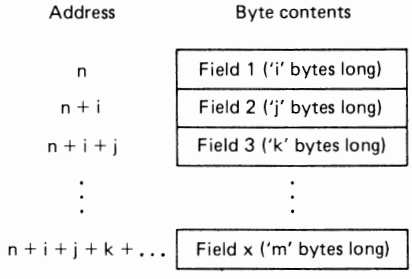


Figure 2.3 Sequential data structure.

Example:

In the record 'mailing_address' we find the fields 'name', 'local_address', and 'zip_code'. A diagram of the complete mailing_address record (Fig. 2.4) shows the order of its fields, starting from the left. The fields in 'mailing_address' would probably be composed of ASCII or XASCII bytes. The data dictionary definition of 'mailing_address' is:

```
mailing_address = name + local_address + zip_code
```

If the length of each field and the record as a whole have been set at a predefined length, the structure shown in Fig. 2.3 is all that is needed. However, if the length of the fields can differ, each field must be followed by an *end-of-field marker*. This marker is a byte value or sequence of values that never occurs as a data value. So if the data are ASCII characters, the value 0 will never occur among them, and it can be used as a marker. Any other noncharacter ASCII value could also be used. An end-of-field marker tells a program where each field ends and the next field begins.

Similarly, if the record length may vary, for instance as fields are added or as fields are removed and the structure is compacted, the record must be followed by

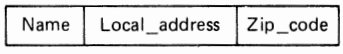


Figure 2.4 Record 'mailing address'.

an *end-of-record marker* having a different value than the end-of-field marker. The end-of-record marker tells a program when it has reached the last field in the record.

Without predefined length or end-of-field and end-of-record markers, a program would interpret all data starting with the first byte of the first field as a single, endless field. Sooner or later the program would "crash."

Selections

There are two important types of selection data structures: the *simple selection* and the *set*. We will discuss these separately.

Simple selections. The simple selection data structure consists of a single data element selected from among several possible elements which are generally of the same type.

The most basic example of this is a selection that can contain either of just two values: true or false. This type of selection structure is called a *Boolean variable*. For instance, a sequence structure covering employee information might contain a copy of a Boolean variable named 'employee present' for every day of the year. Each copy of that variable will have a true or a false value, and thus keep track of the employee's attendance.

More generally, however, the data element will be selected from any finite number of data elements. The symbol showing the selection structure relationship between data elements is a set of brackets enclosing data elements separated by vertical lines. Thus the definition for the selection data structure consisting of three optional elements is of the form

```
data_structure_name = [ element_1_name | element_2_name |
                        element_3_name ]
```

Again, in an actual definition each label would be replaced by a descriptive data name. The order of the elements in a selection definition is unimportant; any one of the elements may be selected during processing.

Selection structures with three or more data elements to select from are usually implemented as single byte variables with unique values for each possible data element. For instance, assume that the selection structure 'family_car' is defined as follows:

```
family_car = [ Chevy | Ford | Chrysler | Volvo ]
```

We might assign the values 0, 1, 2, and 3 to each of the make identifiers listed in the definition. Then by reading the value in the variable we would know the make of the family car.

Sets. We can combine simple selection structures to produce a more flexible type of structure called a *set*. From basic algebra we know that a set is a collection of related items. There can be sets of numbers, sets of automobiles, sets of clothing, and so on. The set containing all items related in any particular way is called the *base set*. A set containing some portion of those items is called a *subset* of the base set. Each item in a set is represented by a data element.

Sets are implemented as a sequence of Boolean variables, one for each data element in the base set. In any given set these variables will have true and false values to indicate which elements are present. So in the full base set all these variables will be set to "true." Subsets of the base set will have true values only for the elements that they contain.

The data dictionary definition for a set structure of three elements is of the form

```
set_name = element_1_name + element_2_name +
           element_3_name
```

where each element has a definition of the following form:

```
element_i_name = [ present | not_present ]
```

A common way of storing sets is to assign each Boolean variable to a single bit, with enough total bytes reserved to contain the entire base set. A 1 value in a bit indicates that the corresponding data element of the base set is present in the particular subset. This structure is illustrated in Fig. 2.5 for a base set of 16 data elements.

Address	Byte contents							
	d7	d6	d5	d4	d3	d2	d1	d0
n	Element 8	Element 7	Element 6	Element 5	Element 4	Element 3	Element 2	Element 1
n + 1	Element 16	Element 15	Element 14	Element 13	Element 12	Element 11	Element 10	Element 9

Figure 2.5 Set data structure.

Example:

Develop a data structure that represents employee attendance during any one week.

We will start by giving the data structure the descriptive name 'week's_attendance'. The structure must identify the days that the employee worked during a 7-day week. This can be done with a base set workweek of seven Boolean variables of type 'workday', where a true value in a given variable indicates that the employee worked that day.

The workday elements are 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', and 'Sunday'. Since the base set contains just seven data elements, it can be represented in 7 of the 8 bits of a single byte.

Assuming that d0 corresponds to 'Sunday' and that the remaining days are assigned to bits in ascending order, the base set looks as shown in Fig. 2.6.

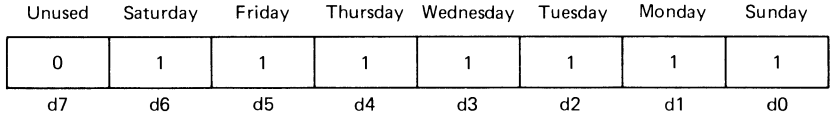


Figure 2.6 Work_week base set.

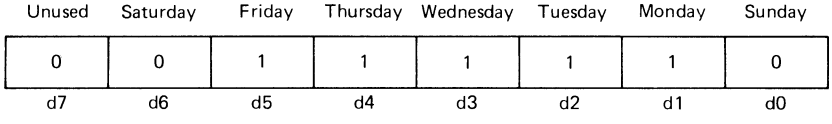


Figure 2.7 Example week's_attendance subset.

If the employee works a full 5-day shift one week, his or her attendance can be shown with the following week's_attendance subset, shown in Fig. 2.7. The data dictionary definition for the week's_attendance set is

```
week's_attendance = Sunday + Monday + Tuesday + Wednesday
                  + Thursday + Friday + Saturday
```

where each day has a definition of the form

```
Sunday = [ present | absent ]
```

A program processing 'week's_attendance' looks at the Boolean variable for each data element in the base type and performs specific actions tailored for those items that are true. For instance, a payroll program may compare an employee's workweek data for the previous week against that employee's usual workweek data and then compute his or her regular, overtime, and holiday pay.

Simple selections and sets do not require terminating markers, since they are always of known and unvarying length.

A program examines a selection or set data structure and then selects what it will do based on what values it finds. This processing pattern can be described as "IF condition is TRUE, THEN do action 1; ELSE do action 2," where action 2 may be another IF . . . THEN . . . ELSE action decision, and so on until all data elements have been tested for and handled. We will see how this works when we discuss assembly language in Chapter 3.

Repetitions

A repetition data structure is composed of repetitions of the same type of data element. The name "repetition" also reflects the repetition of actions performed to process all the data elements in such a structure.

The symbol for the repetition of a data element is enclosing braces preceded by the lowest possible number of repetitions and followed by the highest possible number of repetitions. Omitting the higher number means that there is no upper limit to the number of repetitions (except, of course, the practical limit imposed by the physical capabilities of the computer used). Thus the data dictionary definition for a repetition data structure containing between *n* and *m* repetitions of a data element takes the form

data_structure_name = *n*{ data_element_name }*m*

and the data structure mailing_list, composed of repeated mailing_address elements, would be

mailing_list = 0{ mailing_address }1

Note that preceding the braces with 0 and following them with 1 means that the enclosed data element is optional. An additional symbol that can be used in any structure definition encloses a written description of a lowest-level data item. The symbol consists of parentheses enclosing asterisks enclosing the description. Thus the 'name' field, holding the recipient's name in the earlier mailing_address sequential data structure, could be written

name = 2{ (*ASCII byte*) }13

which defines the 'name' data structure as containing from 2 to 13 ASCII byte values.

There are several variations on the repetition structure. The simplest one places the data elements consecutively in memory. Other types of repetitions are made by combining data elements with pointers.

A *pointer* is a value that points to the location of a data element. A pointer can be the address of the data element, or it can be an address offset into a data area starting at a particular address, or if the data element is one of many consecutive elements having the same length, it can be the position number of the element in the list of elements. We will call these three types of pointers *address pointers, address-offset pointers,* and *position pointers,* and will see how they work as we discuss the various advanced repetition structures.

The different types of repetition structures are discussed individually below.

Simple repetitions. Simple repetition structures are often found in nature. One example is found in flocks of ducks. The structure of a flock is that of many copies of a single element, 'duck'. Note the modeling (i.e., abstraction or data selection) behind our concept of a duck; the unique details of individual birds are omitted so that ducks are in the same category and 'flock' can be easily defined.

A common example of a repetition data structure is a company mailing list, which consists of repeated copies of the element mailing_address. Note that since 'mailing_address' is a sequence, 'mailing_list' is a repetition of sequences. As we have said, data structures consist of elements that can be individual data items or smaller data structures.

A duck hunter "processing ducks" would perform one group of actions on all ducks: "shoot and then retrieve," one duck at a time, multiple times. A program processing a mailing list would also act repetitively in most cases. There would be one group of actions for all mailing_address elements, performed one mailing_address at a time until all mailing_address elements in the mailing_list structure had been processed. The repetition of processing actions is associated with repetition data structures in general and is called a *loop*. Loops are discussed further in Chapter 3. An individual repetition structure is called an *array* or sometimes a *file*.

A file structure whose data elements are one byte long each can be represented pictorially as shown in Fig. 2.8. In general, individual data elements in a file can be longer than one byte; each one can be an entire data structure.

As with the sequence, if the file has a predefined length, the pure structure is all that is needed. However, if the length of the file is allowed to change, the end of the file must be marked. There are two major ways of doing this. First, because all data elements in a file structure are of the same type and therefore the same length, the first element can be preceded by a "number of elements" byte containing a value equaling the number of elements in the current version of the file. A program can use this value to keep track of where it is in the file and to avoid overrunning the end. In the second method, the last element is followed by an end-of-file or EOF

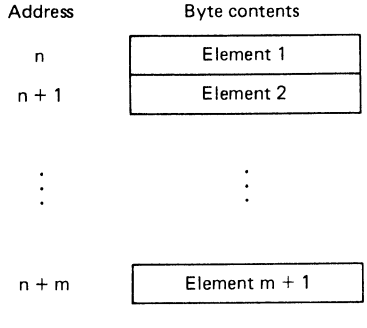


Figure 2.8 File data structure.

marker. As with sequences, the end marker is a byte value or sequence of values that never occurs in a data element.

Either a number-of-elements value or an EOF marker will define the end of a file structure. Which one you choose depends on which matches the intended use of the file best (i.e., which leads to the simplest and most efficient program). However, to make that choice you need programming knowledge that will be provided in the next two chapters.

The next most common type of repetitive data structure is the stack, of which one form was discussed in Chapter 1.

Stacks. The type of stack introduced in Chapter 1 can be extended into a general data structure which is locatable anywhere in the computer's memory. Like the simple repetition, it usually consists of data elements of matching type.

The stack register is too limited for accessing a general stack data structure, so the data elements in a stack structure are accessed with a user-constructed stack pointer. This is the first general use of pointer data we will discuss. A user-constructed stack pointer is almost always an address pointer, made of a two-byte location containing the current address of the top of the stack. A data dictionary definition for this data might appear as follows:

```
pointer_name = (*address*) POINTER TO top_of_stack
```

The general stack structure expands and contracts like the 6510 microprocessor's built-in stack.

Stacks, sometimes called *LIFOs* or *last in, first out* data structures, are useful for organizing any data that should be retrieved in the opposite order to their creation or storage. The stack structure can be represented pictorially as shown in Fig. 2.9.

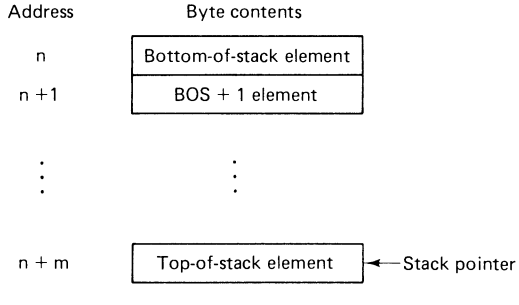


Figure 2.9 Stack data structure.

Queues. A relative of the LIFO is the structure called the *FIFO* or *first in, first out.* FIFOs are also called *queues,* for their similarity to waiting lines such as those at movie theaters and grocery store checkout counters.

When a customer waiting line is first formed it contains no people. As customers enter the line it grows, and as customers are serviced it shortens. People are serviced in the order in which they enter the line. There is little or no connection between the rate at which people are added to the line and the rate at which they leave it.

A FIFO is empty when first created. Data elements are placed in a FIFO and removed from it in an *asynchronous* manner, that is, without requiring that data be deposited and removed at the same time or rate. By definition, data are removed from a queue in the order in which they are placed in it.

FIFOs are useful for data elements that are produced and processed in the same order but not necessarily at the same rate. The Commodore 64 has a FIFO called a *type-ahead buffer,* which accepts up to 10 characters from the keyboard when the computer is busy running a program in BASIC. When the program is finished, the computer's operating system reads the characters and acts on them.

A FIFO is constructed in two steps. First, a memory area of constant length is set aside for the FIFO data structure. The larger the area, the greater the irregularity that can be tolerated in both the input and output data rates. Of course, the *average* rate of removing data from the queue must be greater than or equal to the *average* rate of depositing data on the queue if data are not to be lost.

Second, two pointers are set up: one for the input location in the FIFO and one for the removal location. These are usually either address pointers or address-offset pointers giving the position offsets of the input and output locations from the first location in the set-aside memory area for the FIFO (e.g., an offset pointer to address 8002h in a FIFO starting at address 8000h would have the value 02h).

In use, FIFO is treated much like a circular slide tray, with the last location in the FIFO connected to the first. Imagine that you are visiting a friend who receives a package of developed slides in the mail. On the spur of the moment you both decide to view them immediately. Your friend sets up a slide projector and places an empty circular slide tray on the top. He inserts the first slide into the slot just in front of the projector opening, and you press the button for the tray to advance and the slide to drop in. To make this fiction resemble the operation of a FIFO most closely, we will assume that as the slides are viewed, the slide projector ejects them into a pile on the floor (Fig. 2.10).

As you are advancing the slides, your friend is placing them into the slots in front of the projector opening in the order that they are to be viewed. As long as the slides are being viewed and ejected at least as fast as they are placed in the tray, there are no problems. Of course, you may have to wait for your friend to insert a slide if you catch up with him. However, if your friend fills the tray while you are viewing a particularly interesting shot, he must wait until you begin ejecting slides again before

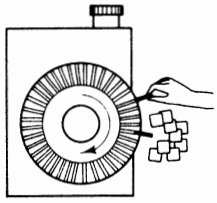


Figure 2.10

he can enter them again. Otherwise, he will jam the projector by placing slides into locations that already hold slides.

In a FIFO data structure, the location from which data will be removed corresponds to the slide tray slot over the projector opening. As each data element is removed, the output location moves to the next memory address, just as the source of the next output slide moves to the next tray slot. Similarly, input data items are placed into consecutive memory locations just as new slides are placed into consecutive slots.

If the input and output locations are constantly changing, how can those locations be tracked? In a slide tray, the human user keeps track of the current input slot by sight while the projector automatically tracks the current output slot. In a data FIFO, the pointers keep track of the input and output locations for the program using that structure. In a data dictionary, the pointer data could be defined as follows:

```
pointer_name = (*address-offset*) POINTER TO FIFO_element
```

The FIFO structure and its supporting pointers can be represented pictorially as shown in Fig. 2.11. The circular interpretation of the FIFO means that address n is considered to follow address n + m.

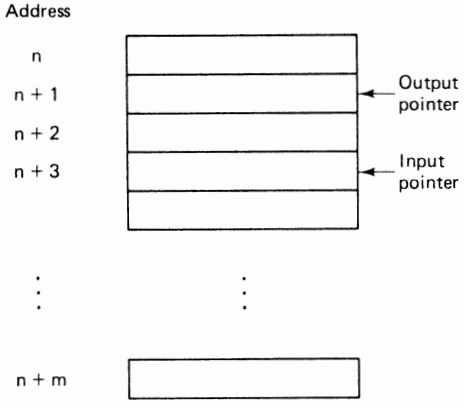


Figure 2.11 FIFO data structure.

When the FIFO is first created, the input and output pointers both point to location n. The FIFO is not activated until the first data element is placed into n and the input pointer is incremented to point to n + 1. Removing a data element causes the output pointer to be incremented, so that the output pointer effectively "follows" the input pointer.

As long as the input pointer stays ahead of the output pointer without catching up to or lapping it, data can be input and output without any delay. However, if the input pointer catches up with the output pointer, it means that all the FIFO locations have been filled with input data. Further input must be refused until FIFO locations have been emptied by output operations. Alternatively, if the output pointer catches up with the input pointer, that is, if the FIFO is empty, there can be no further output until input data have been received by the FIFO.

Linked lists. Another type of data structure results from pairing a pointer with each data element in a simple repetition, where the data elements are in some logical order, such as alphabetical or numerical. If the pointer attached to each data element points to the next element in the logical order, the resulting structure is called a *linked list*. We will use *position pointers* in our linked lists, although other types will do just as well. Such a pointer is defined in a data dictionary as follows:

```
pointer_name = (*position*) POINTER TO list_element
```

Linked lists are used because they can change their length. Therefore, a linked list needs an end-of-list marker to notify a program when it has reached the last data element. This marker is an impossible or *nill* value placed in the last element's pointer. If position pointers are used and the first data element is defined as being at position 1, the value 0 will never be used and can be defined as the *nill pointer*. Further, certain 6510 instructions that we will study in Chapter 3 make the value 0 particularly easy to detect and react to.

It is difficult to give a totally general representation of a linked list: The physical order of the elements can vary from their logical order in any number of ways. So we will show a specific linked list that illustrates the linked-list concept adequately. The linked list shown in Fig. 2.12 contains four elements. Each element contains a one-byte data element and a one-byte pointer. The element numbers reflect their *logical* rather than *physical* order. The pointer numbers reflect the physical position of the next element in the list (e.g., a pointer number of 3 points to the third element in the list). In general, of course, the data elements can be longer than one byte.

One advantage in having a pointer with each data element is the ease with which the logical order can be changed. To insert a new element in the logical order, for instance, the new element is first placed in an unoccupied physical space in the list. Since a list usually fills a contiguous memory area, this would be the first address following the last list element. Then the data element's proper logical position

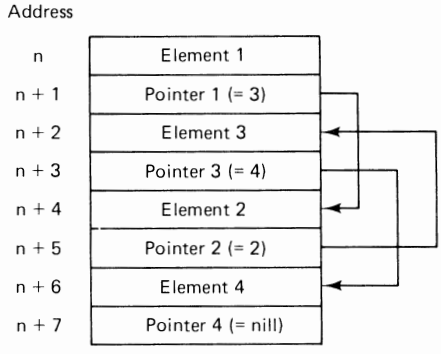


Figure 2.12 Linked-list data structure.

in the list is located. Finally, the affected element pointers are changed to place the element in the proper logical position. The stages of this process are illustrated in the following example.

Example:

There is a linked list of names in alphabetical order. Initially, the list contains only the names 'Franklin' and 'Smith'. Insert the name 'Jones' by changing the name pointers.

As shown in Fig. 2.13, the JONES element is added to the list by transferring the value in the FRANKLIN pointer (i.e., 2 if position pointers are used) into the JONES pointer so that JONES will point to SMITH. Then the FRANKLIN pointer is loaded with the value of the position of the JONES element in the list (i.e., 3) so that FRANKLIN will point to JONES. The order of the list is then FRANKLIN, JONES, SMITH. Reversing this process, the name JONES can be deleted by moving the value in the JONES pointer into the FRANKLIN pointer. Then FRANKLIN points to the element that previously followed JONES, and there are no pointers to JONES, so it has effectively been deleted.

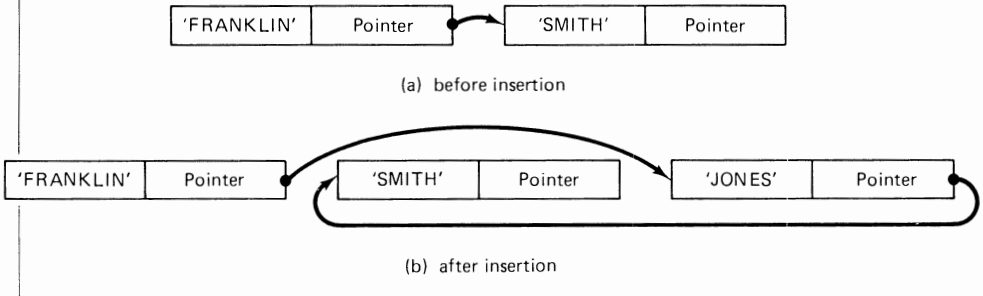


Figure 2.13

If the pointer of the last element in the list points to the first element in the list, the structure is called a *ring*. By definition, a ring has no end-of-list marker. In a ring, data elements earlier in the logical order can be reached from elements later in the order by continued forward movement through the list via the pointers.

Another variation on the linked list is made by attaching *two* pointers to each data element, one pointing to the following element and one pointing to the preceding one. The additional effort needed to process two pointers in operations such as insertions and deletions may be offset by the advantage of being able to back up in the list while searching through it.

Hierarchies. Pointers can also be used to indicate family relationships among data elements. The structure based on this use of pointers is called a *hierarchy* or a *tree*.

Each data element in a hierarchy is accompanied by two or more pointers to directly related data elements. First, a central or *root* data element points to two or more directly related data elements. Then each of these elements points to two or more elements directly related to it. This *branching* continues down to the last elements in the structure.

A common hierarchical structure is a company's organization chart. The name of the president of the company is the central element in the hierarchy. It points to the names of the company vice-presidents, the names whose roles are the most closely related to the role of the president. This structure typically continues down to the level of the individual supervisors or even the worker employees.

As with the linked list, it is difficult to give a general representation of all hierarchies. The nine-element hierarchy shown in Fig. 2.14 illustrates the structure of hierarchies in general. Each element is represented by a box, and elements with family relationships are connected with lines. We will see shortly how the boxes and lines are implemented.

The following example shows a hierarchical structure of family names.

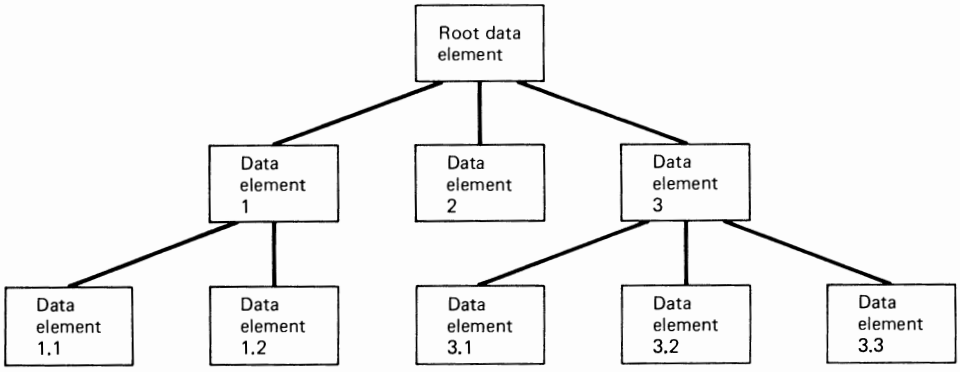


Figure 2.14

Example:

Show three generations of an imaginary family's lineage in a family tree.

In this example each element (i.e., name) is dependent on the names of the immediate ancestors. These relationships are the most direct and important to a family tree, so elements so related are linked directly (Fig. 2.15). Indirect relationships such as cousin or uncle can be found by tracing through multiple elements and applying rules concerning relative levels in the tree.

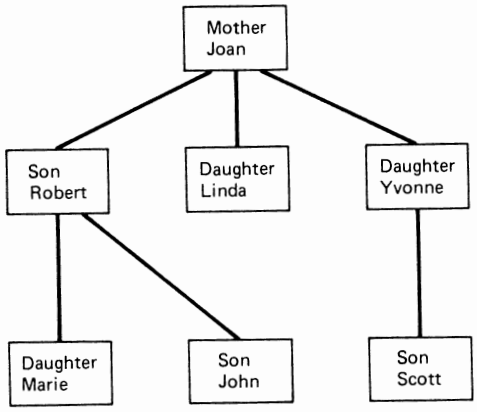


Figure 2.15

The parts of a hierarchical data structure are named according to the tree metaphor. As you have seen, the top data element is called the *root*. The lowest data elements are called *leaves,* and data elements on intermediate levels are called *branches.*

Only downward pointers are needed in a hierarchy. Each data element has one pointer for each attached element on the level beneath it. Even though different data elements can have different numbers of branches downward, to simplify processing the data structure, all elements should be given the same number of pointers. Unused pointers in a given element, or the pointers for an element on the bottom level of the hierarchy, can be given a nill value. The number of pointers should equal the maximum number of branches used by any element in the hierarchy. The data dictionary definition for a data element in a tree that allows up to four child data elements per parent element is as follows:

```
tree_element_name = data_structure_name + pointer_1_name +
                    pointer_2_name + pointer_3_name +
                    pointer_4_name
```

with the pointers defined elsewhere in the data dictionary in the usual manner. This structure is shown pictorially in Fig. 2.16.

A hierarchy breaks data into *categories* according to which branch they belong to. Data whose categories and therefore branches can be calculated beforehand are the most natural candidates for placement in a hierarchical structure.

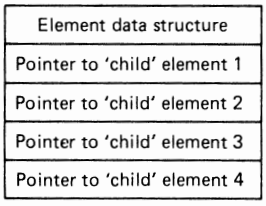


Figure 2.16 Hierarchical data element.

Example:

Identify an everyday collection of information that can be organized hierarchically.

A cookbook contains information of this type. The main branches in a cookbook hierarchy might be to national groupings such as 'Mexican' and 'French'. The next-lower branches could be to main ingredients such as 'Fish', 'Poultry', 'Beans', and so on. Finding a recipe would be a matter of traveling down a limited number of category branches, instead of searching linearly through an unpredictable number of elements in a file structure.

Example:

Develop a data structure for storing data records for the members of an automobile club.

A logical way to organize member information is by the type of car owned. A hierarchical data structure whose top element is 'club membership' can be broken into 'auto type' data elements underneath. These can then be broken into the individual member records. With this structure, finding the names of all the members owning RX-7's, for instance, is simply a matter of retrieving all the names in the RX-7 branch.

For data that can easily be categorized, especially if such data are in large amounts, hierarchical organization can allow using the fastest and most efficient accessing methods in your programs. Indeed, a whole class of programs called *data base managers,* which are used to catalog and manipulate data flexibly, favor two structures for data storage. One of those structures is the *relational data base;* its study is outside the scope of this book. The other structure is the hierarchy.

THE VIRTUE OF SIMPLICITY

The simplicity of programming is directly affected by the simplicity of the data structures used. No matter how complex and expansive the data are that must be handled to perform a task, those data can be organized as levels of the simple data

structures we have discussed. Data dictionary definitions for such structures can be built from combinations of the basic symbols for the simpler structures being combined.

Example:

Show the data dictionary definition for a structure built by combining the simple structures.

The zip__code field in the mailing__address sequential data structure has a combination structure of this type. It consists of a selection of two possible repetition structures, as seen in the following definition:

```
zip__code = [ 5 {(*ASCII digit*)} 5 | 9 {(*ASCII digit*)} 9 ]
```

which means that zip__code is composed of either five or nine repetitions of an ASCII digit. The choice allows for either five- or nine-digit zip codes in an address.

After data have been structured in this way, program processes can be designed to transform each basic structure or level of basic structures. This keeps to a minimum the portion of an information-handling task being dealt with at any given time. The simple regularity of the data structures discussed in this chapter leads to simple and regular programs, which is one of the most important goals of the programming craftsman.

Exercise:

Review what you have learned by answering the following questions.

(a) What are the two stages of modeling called, and what is done in each stage?

(b) Describe the characteristics of each of the three basic types of data structures.

(c) What is the difference between a LIFO and a FIFO?

(d) Think of a category of information whose data might best be stored in a hierarchy.

FOR FURTHER STUDY

Subject	Book	Publisher
Data structures	*Algorithms + Data Structures = Programs* by Niklaus Wirth	Prentice-Hall, Inc.

INTO ITS BRAIN: 6510 ASSEMBLY LANGUAGE

Achilles, the noble but tragic figure in Homer's the *Iliad,* deeply resented the king of his native Greece. With the Trojans counterattacking, Achilles withdrew from the defense of his countrymen. His close friend Patroclus borrowed Achilles' armor and drove the battle away from the Grecian ships. When Patroclus was killed and the armor fell into Trojan hands, Achilles in grief and guilt determined to avenge his friend. The craftsman-god Hephaestus favored Achilles and forged the metals copper, tin, gold, and silver, hued by sulphur, into an iridescent armor for Achilles' defense.

The eighth century B.C. account in the *Iliad* is one of the earliest records of the art of coloring metals by reacting them with sulphur or arsenic. Many civilizations equated such natural transmutations with the more significant transmutations of sickness to health and poverty to wealth. This chemistry evolved into the practice known as *alchemy.*

Alchemy has been practiced by the Chinese, the Indians, the Greeks, the Arabs, and the mediaeval Western world. The Greek, Arab, and Western alchemists concentrated on chemically transmuting the base metals to gold. In the nineteenth century the chemical production of gold from base metals was finally proven impossible, although Adolf Hitler and others have attempted it since.

Alchemy was important to the ancients because most of their civilizations treated precious metal as the most valuable commodity on earth. As civilization progressed, information revealed itself as an even more valuable commodity. Unlike metal, however, information in the form of data is easily transmuted in both content and structure to increase its inherent value. Data transformation is an alchemy of greater potential profit than the alchemy of minerals ever could have been.

In Chapter 2 we referred to the CPU as a transmutation device. It is the modern alchemical engine. To carry out its function, the CPU has a complete repertoire of operations that a computer program can activate in any desired order. CPU operations and their control with computer programs are the main topics of this chapter.

COMPUTER PROGRAMS

A computer program is a collection of instructions, each of which activates one or more CPU operations, with the collection as a whole performing an information-handling task. Programs come in two forms: the written form that the programmer works with, and a machine form, either in the directly executable operations code of Chapter 1 or in a code that an interpreter program running in the computer at the same time can easily convert to the operations code for step-by-step execution. The written form of a program is called *source code.* The executable operations-coded form of a program is called *object code* or *machine language code.*

A *programming language* is a set of rules that defines a format for written programs. Thus programming languages exist for the convenience of the programmer, not of the computer. The key elements of a language format are a set of individual instructions complete enough to allow programs built with them to perform useful tasks, and a *syntax* or *grammar* governing the allowable relationships between those instructions (e.g., their order, grouping, etc.).

There are two major types of programming languages: *high-level languages* or *HLLs,* and *assembly language.* Popular HLLs include BASIC and Pascal. The main difference between HLLs and assembly language is that an HLL instruction corresponds to many CPU operations, whereas an assembly language instruction corresponds to just one CPU operation. Thus an HLL instruction might be "write the following sentence on the TV screen . . .," which will be translated into many object code instructions, while an assembly language instruction might be "load the byte at address 8000h into the accumulator," which will be translated into only one object code instruction.

The "one-to-many" translation of HLL instructions into CPU-executable object code is performed by programs called *translators.* One class of translators converts the source code into object code as a whole, with the object code used from that time on. This type of translator is called a *compiler.* Another class of translators converts the source code into object code one source instruction at a time and executes the translated code immediately; the source code is always used. This type of translator is called an *interpreter.*

All translators are "automatic programmers" in that they select a combination of CPU-executable operations to perform each HLL instruction. However, program instructions are not islands; they exist in a context of other program instructions. A translation that is tailor-made in one context may not work at all in another. There are so many possible contexts that in practice, translators must either

catalog a few major contexts and their best compromise translations, or worse, use entirely safe but even less efficient "universal" translations for each instruction. Most BASIC translators come under the latter category. Inefficient translation is the main reason for the speed and size disadvantages of HLL programs. Further, BASIC programs are kept as source code and executed under the direction of an interpreter, which slows them even more.

The advantage of HLLs, of course, is that the programmer writes shorter programs at a higher logical level. Given the memory and speed limitations of personal computers, however, the disadvantages of HLLs usually outweigh the advantages for performing significant tasks.

The one-to-one conversion of written assembly language instructions into object code instructions is performed by programs called *assemblers.* Since the CPU operations used to perform any task are hand-picked by the programmer, a program's efficiency is limited only by the programmer's knowledge and experience. The necessary knowledge for efficient programming is provided in the next two chapters. Gaining experience is up to you, but even an inexperienced programmer can create far more efficient programs in assembly language than is possible with HLLs.

Our next topic is the programming language aspect of assembly language. We will learn how assembly language instructions are formatted in programs. Then we will explore the instructions themselves and the CPU operations they represent. Next we will see how instructions are combined to do useful work on the various types of data structures. Finally, we will see how to improve and correct programs to obtain the best possible program performance.

ASSEMBLY LANGUAGE

As you may recall from Chapter 2, each object code instruction consists of two parts: an operation byte, which the CPU decodes to determine what it is to do, and between zero and two operand bytes, which hold or point to any data on which the CPU will perform the operation. An operand byte consisting of data for the operation to manipulate is called a *data operand.* An operand consisting of an address pointer is called an *address operand.*

In assembly language, each CPU operation is represented by a three-letter abbreviation. Operands can be represented with programmer-assigned names or with written numbers.

Operation abbreviations are called *mnemonics,* which literally means "memory aids." For instance, there is a CPU operation that loads a data byte into the accumulator. This operation has been given the mnemonic LDA, which means LoaD the Accumulator. Every CPU operation has its own mnemonic, giving the programmer complete control over the CPU.

As we said, operands can be represented with written-out numbers such as "82," or with programmer-assigned descriptive names such as "count." Such

names are symbolic representations of a number. Like mnemonics, they make a program easier to read, write, and correct. However, not all assemblers allow the programmer to define such symbols; a restricted type of assembler called a machine language monitor does not, for instance. Even when using a machine language monitor, however, your handwritten or typed-in copy of a program should use symbols for understandability.

The Commodore assembler limits symbols to six alphanumeric characters in length, where the first character must be a letter and the remaining characters may be letters or numbers. Other assemblers may allow fewer or more characters.

The simplest type of address operand was described in Chapter 1. It consists of the actual address of the data to be manipulated. Other types of address operands allow for CPU changes to the address value before it is placed on the address bus. Each different type of address operand is matched to a particular type of data structure. The careful use of these addressing methods is the first key to effective assembly language programming.

The addressing method used with an address operand is indicated to the assembler by accompanying the operand with assembler-defined emblems such as # and ( ). For instance, # temp could be one such operand. We will postpone defining these accompanying emblems until the addressing methods have been discussed.

Operands must have their number code defined to the assembler with accompanying emblems. This avoids any ambiguity with numbers such as 10, which could otherwise be interpreted in binary, decimal, or hex code. Following are the number code tokens recognized by the Commodore assembler:

Emblem	Number code	Example
None	Decimal	37
Prefix $	Hexadecimal	$FA
Prefix %	Binary	%00010101
Enclosing apostrophes	ASCII	'try again'

In a symbolic assembly language program, names are assigned their numeric values with *assignment directives*. Directives are special instructions to the assembler which are inserted in the source code. Assignments are a type of directive which consists of a symbolic name, an equals sign, and a numeric value. For instance, "home = 128" assigns the value 128 decimal to the word "home."

The assembler processes each assignment directive during assembly and places the symbolic name and its corresponding value into a *symbol table*. For the rest of the assembly, every time a name is encountered in the source code it is searched for in the symbol table, and, if found, its corresponding value is substituted in the resulting object code. So an assignment must always precede the first use of a symbol. Directives are used only during the assembly process; they have no counterpart in the object code.

The format of an assembly language instruction typically includes places or

fields for the operation byte or *opcode,* the operand, a comment, and a symbolic name or *label* for the address of the operation byte in the object code. The instruction format for the Commodore disk assembler appears as follows:

```
(label)  (opcode)  (operand)  (comments)
```

where the parentheses mean that the field is optional. The first address in the machine-executable program is defined in the source code with a special directive. Most assemblers require this assignment at the very beginning of the program, so that all address operands and symbolic data can be given known address values during assembly.

The opcode field holds a mnemonic. The operand is a data byte or an address in numeric or symbolic form, with possible accompanying symbols to indicate addressing method or number base. Comments should be preceded by a semicolon for clarity and to allow a comment to take a line by itself.

We can make the following instruction using the mnemonic LDA:

```
movbyt  LDA #temp   ;move the constant value 'temp' into A
```

Normally we would omit such a trivial comment, however.

All the examples in this book will be in the Commodore assembly language format. The format of other symbolic assemblers is so close that translation is trivial.

Good assemblers and machine language monitors are available from Commodore and HES. MindTools will be selling a complete and inexpensive assembly language programming system, based on the Power Programming method, by mail through major C64 magazines in early 1986. You will need an assembler or monitor to practice the principles in this book. The assembler's accompanying manual will describe the specifics of its use. Assemblers find certain syntactic or formatting errors for you during the assembly process. They also usually include a debugging program for finding semantic, or logical, errors while the object program executes. The assembler manual should also describe these features.

The two elements of an assembly language instruction, the operand and the operation, can now be studied. Accessing data is the most basic aspect of any microprocessor's operation, and therefore of assembly language. Therefore, the operand and its associated addressing methods will be our first concern. After a brief review and discussion of addressing in general, we will separate the addressing methods into categories based on the type of data structure each is best at accessing.

Addressing

The 6510 has two main methods for addressing data. In the first method, the CPU obtains a pointer to data and uses it without modification to access data. The pointer is usually an address operand. This method is called *direct addressing.*

In the second addressing method, the CPU obtains a pointer and adds to it a

byte called an *index* to get a final data address. This method's pointer is also usually an address operand. This method is called *indexed addressing*. The data registers X and Y are the two possible index sources.

These two addressing methods have very different strengths. Direct addressing is best suited to accessing sequential data structures. Indexed addressing is best with repetitive data structures. Both methods are useful with selective structures. The reasons for these affinities will become apparent shortly.

Each addressing method allows for several addressing *modes* or submethods. Modes allow the accessing method to be fine-tuned for the specific data structure being processed.

The addressing methods and modes will be discussed in terms of the data structure types they access best. Later in this chapter you will be using these modes in combination with the CPU operations to perform small but interesting tasks on your computer. The programming of larger tasks requires techniques that will be explained in Chapter 4.

Sequential data. Recall that a record is a collection of dissimilar field data elements. For fast instruction execution and simplicity, data in small records should be directly accessed with unaltered operands (i.e., by direct addressing).

The direct addressing modes are optimized for accessing two special cases of the record structure: the single-byte constant and the small record of single-byte variable fields. With larger records or larger variable fields the suitability of direct versus indexed addressing will have to be judged on a case-by-case basis. You will be ready to make such judgments by the end of this chapter.

Single-Byte Constants. By definition, a constant is a data value that never changes. To a program, the main difference between constants and variables is that constants can be placed in both changeable and unchangeable memory locations, while variables can be placed only in changeable locations. Since a program's instruction bytes are unchanging, constants can be stored and accessed as operands, whereas variables cannot. Variables can be stored in RAM or in registers and are accessed with unchanging pointers in the program instructions.

Placing a constant in an instruction saves program space and execution time. Space is saved since there is no extra pointer to store. Time is saved since the fetching of an address operand and its placement on the address bus is eliminated.

A constant data operand is called an *immediate operand*. An instruction containing an immediate operand fills two bytes in the memory map. Graphically, this appears as shown in Fig. 3.1.

Accessing data in an immediate operand is called *immediate addressing*. The execution of an instruction using immediate addressing is as follows. The opcode is fetched and decoded, the PC is incremented, and the immediate operand is retrieved and processed. The PC is then incremented again to point to the next instruction. Thus in this addressing mode the PC serves as the data pointer. Most of the other direct modes use address operands as pointers.

Earlier we mentioned the emblems that indicate the addressing mode of an

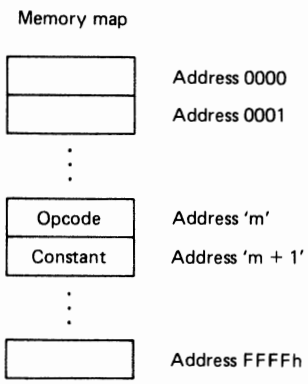


Figure 3.1 Immediate addressing

operand. The emblem for the immediate addressing of an operand is a # as the first character in the operand.

Example:

The 6510 operation called LDA has already been mentioned. One of its addressing modes is immediate addressing. Show the full instruction for moving the constant value 1Dh into the accumulator, using immediate addressing. Then show the same instruction with the value 1Dh represented by the symbol 'temp'.

Combining the LDA mnemonic with the immediate operand yields the instruction

```
LDA #$1D
```

(Recall that assembly language uses the $ sign to indicate the hex number base.) Substituting the symbol 'temp' for the number in the instruction above produces

```
LDA #temp
```

The latter form illustrates how symbols can improve the understandability of assembly language instructions.

Small-Record Variables. A small record of single-byte variables should normally be accessed with unaltered pointers to nonprogram memory or CPU register locations. Such pointers are found in all the remaining direct addressing modes. They will all be discussed in this section.

As with constant data, the placement of variables determines the method and efficiency of their accessing. The most general placement for a variable record is anywhere in RAM. Accessing a byte in such a record requires a 16-bit address operand.

A 16-bit address operand is called an *absolute address.* An instruction containing an absolute address fills three bytes in the memory map. The address operand

points to the data to be processed by the opcode. Graphically, this appears as shown in Fig. 3.2.

Using an absolute address to access a data byte is called *absolute addressing*. The execution of an instruction with an absolute address proceeds as follows. The opcode is fetched and decoded, the PC is incremented, the low-order byte of the address operand is retrieved and placed in LOBUF, the PC is incremented, the high-order byte of the address operand is retrieved and placed in HIBUF, the address buffer is placed on the address bus, and the data byte is retrieved and processed. The PC is then incremented to point to the next instruction.

No addressing mode emblems are used with absolute addresses, since the assembler can determine the addressing mode from the context of the instruction.

Example:

Show the assembly language instruction for loading the acculumator with a data byte at memory location 8D7Fh. Show the same instruction with the symbolic label 'text' representing the data address.

Combining the LDA mnemonic with the absolute address yields the instruction

```
LDA $8D7F
```

Substituting the symbolic label 'text' results in the instruction

```
LDA text
```

By restricting the placement of a record to locations entirely within *page-zero memory* (i.e., between addresses 0 and FFh), only one byte is needed for a pointer.

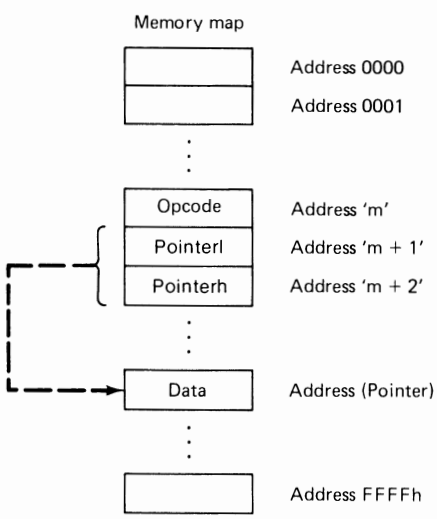


Figure 3.2 Absolute addressing

This shortens the instruction length by one byte and instruction execution time by one byte retrieval. Execution speed is so improved that page-zero memory is often treated as a bank of fast-access registers such as those in the 6510. Using an 8-bit address to access a data byte is called *zero-page addressing*.

Example:

Show the instruction that loads the accumulator with a data byte stored in address 7Fh.

Combining the LDA mnemonic with the zero-page address produces the instruction

```
LDA $7F
```

By restricting the length of a variable record to one byte and by limiting its placement to the internal registers of the 6510, the address pointer can be eliminated. Instead, the opcode byte "implies" the registers to be accessed. When the 6510 decodes the opcode, it performs the specified operation on the data in the implied registers. Hence this mode is called *implied addressing*. Only a few 6510 instructions allow implied addressing. For those that do, instruction size and execution time are reduced to an absolute minimum.

Pointers in the foregoing three addressing modes point to variable data. In yet another addressing mode, the pointer contains the address of another pointer in memory. With some microprocessors this secondary pointer holds the address of a data byte. However, in the 6510 direct addressing method the secondary pointer holds the address of a program instruction. This use of the secondary address will be explained later in the discussion of the JUMP operation. In the 6510 both addresses are 16 bits long. A pointer to a pointer is called an *indirect address*. The data addressing use of an indirect address is shown in Fig. 3.3.

The mode using indirect addresses is called *indirect addressing*. It can be thought of as two consecutive executions of simple direct addressing: The address operand is placed on the address bus to retrieve the secondary address through the data bus, and the secondary address is placed on the address bus for the retrieval of the data byte. Although the 6510 does not implement this process in its pure form, it does support two indexed variations on it. We will consider them shortly.

Table 3.1 summarizes the direct addressing modes available for accessing

TABLE 3.1 DIRECT ADDRESSING MODES

	Data type	
Record length	Constant	Variable
1 byte	Immediate	Zero-page *Implied Absolute
2+ bytes	*Zero-page Absolute	*Zero-page Absolute

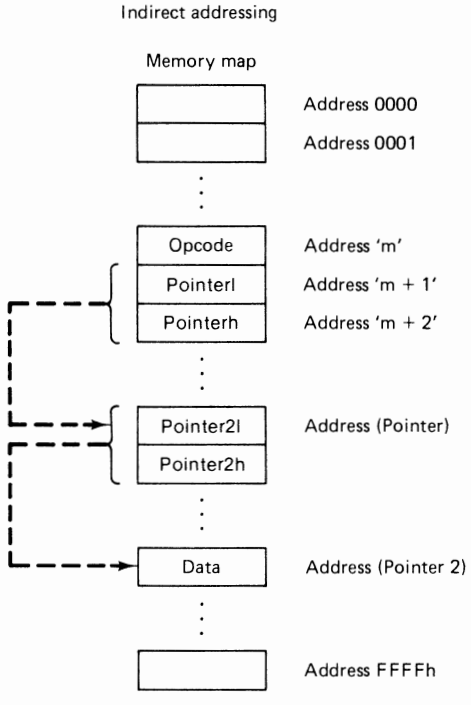


Figure 3.3

records of constants and variables. An asterisk highlights the fastest executing addressing mode for each combination of data type and structure length.

Look over the table now as a review of the uses for the direct addressing modes. Later, it and its counterpart in the repetition section will help you write the most efficient form of an instruction for a given data structure.

Exercise:

Try answering the following questions.

(a) What is direct addressing?

(b) Describe the absolute addressing process.

(c) How do zero-page and absolute addressing differ?

(d) Describe the immediate addressing process.

Selection data. Recall that a set is a group of Boolean or true/false elements, often implemented as one or more bytes containing one Boolean value per bit. Sets can contain all-variable or all-constant set elements. Variable sets can be stored only in memory or register locations, whereas constant sets can be kept in either of those types of storage as well as in immediate operands within a program. Sets in memory can be processed one byte at a time within memory, or placed in the accumulator one byte at a time and then processed. The latter choice is usually best,

since the bit values can be tested so much more quickly in the accumulator that the time spent loading the bytes from memory is regained several times over.

Single-byte variable sets in memory should be accessed with an address operand mode of direct addressing. Multiple-byte sets stored in memory should usually be loaded into the accumulator with the indexed addressing modes, which will be discussed in the next section. Sets stored as constant operands are of course loaded with the immediate mode of direct addressing. Once set elements are in the accumulator they can be accessed with the implied mode of direct addressing, that is, with operations that assume the data object is in the accumulator.

A summary table for this section would simply list the direct and indexed addressing modes discussed in the bracketing sections, so it will be omitted.

Exercise:

Review this section by answering the following questions.

(e) In what location should set elements be placed to be manipulated?

(f) Under what circumstances should direct addressing be used on sets?

(g) Under what circumstances should indexed addressing be used on sets?

Repetition data. Recall that a repetition data structure is a collection of data elements of the same type. Because such structures are usually processed by stepping through their elements, a useful addressing method allows the defining of a beginning or base address for the structure, and the quick manipulation of an offset or index into the structure. The addition of a base address with an index to create a final address defines the indexed addressing method. The modes of indexed addressing are optimized for accessing array and stack data structures.

Arrays. In most situations, each element in an array will be processed the same way as all the other array elements. To keep the number of program instructions at a minimum, each instruction accessing the array must be able to point to every array data element.

The direct addressing modes are useless in this situation. With their unalterable data or pointer operands, most direct modes tie an instruction to a single data byte or address. The indirect addressing mode allows for changing the final location being accessed, but the 6510 uses this mode for purposes other than data addressing. Even if indirect data addressing was provided, repeatedly changing a secondary pointer would be an awkward way to access an array.

Indexing solves this problem. The base address can be obtained using any one of three direct addressing mechanisms. The index can be obtained only from a CPU register, where it is kept for easy alteration to point to different array locations. Depending on the mode of indexed addressing, either the X or the Y register will hold the index value.

Variety in indexed addressing comes from the different ways of obtaining the base address. The three modes of direct addressing used to generate a base address are the absolute, zero-page, and indirect modes.

As with sequences, the placement and size of a repetition determines the method and efficiency of its addressing. If the array is to be stored in memory at large, the base address can be generated with the 16-bit address operand of the absolute mode. The index can be taken from the X or Y registers, at the programmer's choice. Combining a 16-bit address operand with an index produces the *absolute indexed* modes. These modes are known as *absolute X* and *absolute Y,* depending on the register providing the index. Their memory requirements and effect during execution are shown in Fig. 3.4.

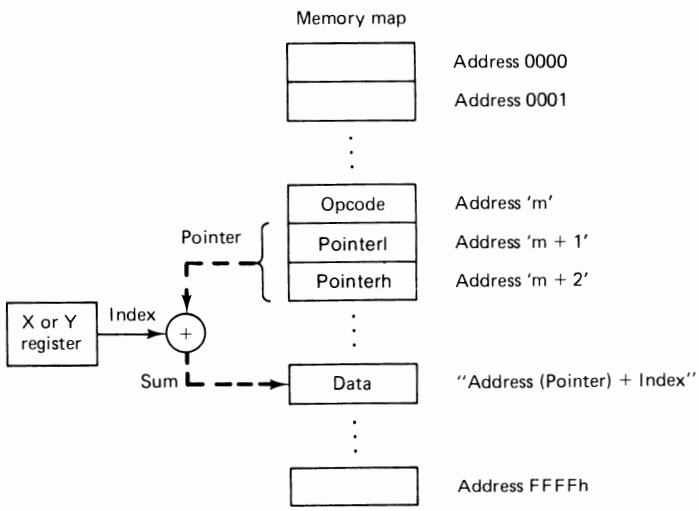


Figure 3.4 Absolute indexed addressing

The execution of an instruction using an absolute indexed address is as follows. The opcode is fetched and decoded, the PC is incremented, the low-order byte of the address operand is retrieved and added to the index, the result is placed in LOBUF, the PC is incremented, the high-order byte of the address operand is retrieved and adjusted if there is a carry from LOBUF, the result is placed in HIBUF, the address buffer is placed on the address bus, and the data byte is retrieved and processed. The PC is then incremented to point to the next instruction.

Absolute indexed addressing is indicated by an absolute address followed by a comma and the register source for the index.

Example:

Show the instruction for loading the accumulator with the value at the address produced by adding the base address 8D7Fh with the index value in register Y.

This instruction is

```
LDA $8D7F,Y
```

If an array can be placed entirely within page zero memory, a zero-page address operand can be used to save time and space. The zero-page indexed modes are called *zero-page X* and *zero-page Y,* depending on the register source of the index. Other than the length of the address operand, the zero-page indexed modes work just like the absolute indexed modes.

Example:

Show the instruction for loading the accumulator with the value at the address produced by adding the base address 7Fh with the index value in register X.

This instruction is

```
LDA $7F,Y
```

There are two situations where absolute and zero-page indexing are insufficient. First, if the array is longer than 256 bytes, the byte-wide index combined with a constant base address cannot access all array locations. Second, if multiple arrays of the same type are being processed in the same way, the previous addressing modes prevent using the same program instructions to access them all.

Both situations can be accommodated with a changeable base address. Both absolute and zero-page addresses are unchangeable. However, the only unchangeable address in the indirect mode is the pointer to the secondary pointer. The secondary pointer resides in general memory, so it can be changed. There is an indexed addressing mode that uses the secondary pointer as its base address, so that both the index and the base address can be adjusted. Thus arrays larger than 256 bytes, and multiple arrays in different locations, can be accessed in a thoroughly general manner.

The price for this flexibility is time. Retrieving an address operand, then a secondary address, indexing, and finally retrieving a data byte is time consuming. In the two situations listed above, however, the natural fit of this addressing mode to the data probably saves more execution time for the overall program than it loses.

To save some of the time taken by indirect addressing, the primary pointer is limited to 8 bits length. Thus the secondary pointer must reside in page zero memory. A further limitation on this mode is that the index value can come only from register Y. Appropriately, this mode is called the *indirect indexed* addressing mode. It is illustrated in Fig. 3.5.

The emblem for an indirect operand is enclosing parentheses, as is shown in the following example.

Example:

Show the instruction that loads the accumulator with the data byte pointed to by the address in zero-page locations addr and addr + 1, indexed by the contents of register Y.

This instruction is

```
LDA (addr),Y
```

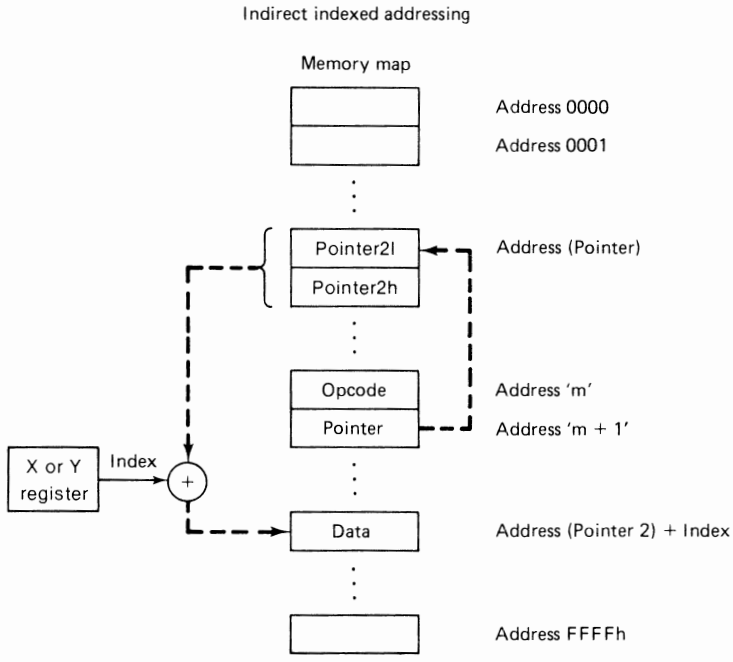


Figure 3.5

An array of 16-bit address registers can be created with the last array-addressing mode. The three previous indexed modes obtain a base address and add an index. This fourth mode uses the index to obtain a base address. It is descriptively called the *indexed indirect* mode.

Indexed indirect addressing starts with a zero-page address operand, which points to an array of 16-bit addresses in page-zero memory. An index stored in the X register is added to the zero-page pointer to form the final pointer into the array. If the sum of the index and the original pointer is greater than 255, the resulting address wraps around to the low end of page-zero memory. So if the sum would be 102h, the new pointer address becomes 2h, and so on. The final pointed-at location, an address element in the array of addresses, is retrieved and used to access a data byte. Indexed indirect addressing is illustrated in Fig. 3.6.

Indexed indirect addressing allows many addresses and individual data elements to be selected from easily. As such it may be more useful for accessing fields in a record than for accessing repetition data structures. For instance, the beginning addresses of each field in a record of different-size fields could be handled with this mode.

Since indexing is used to access the address in page zero, it is unavailable to step through elements in an eventual target array. However, the address in the page-zero array can itself be incremented or decremented to step through a data array, although with a large time penalty.

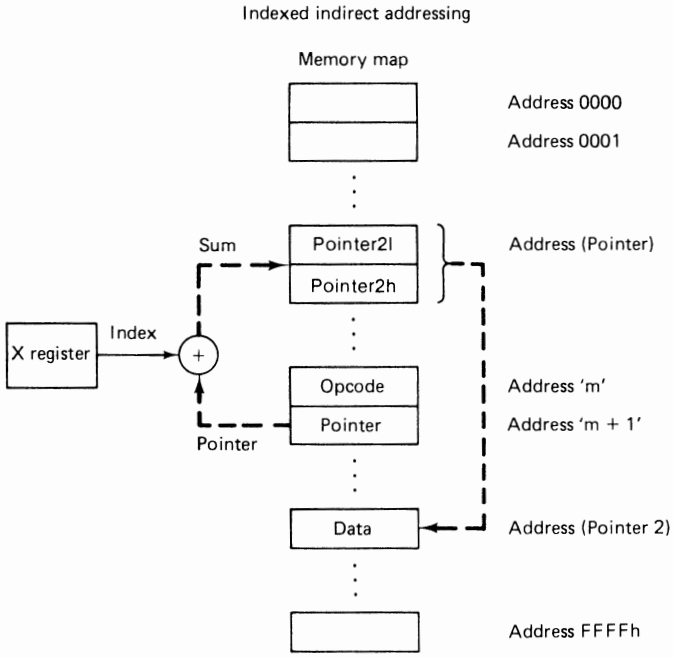


Figure 3.6

Indexed addressing generally adds one tick of the system clock to an instruction's execution time. With instructions using a direct address mode averaging four 1-microsecond ticks in length, the penalty is a 25 percent slowdown. As we have pointed out earlier, though, the alternative in using direct addressing is usually worse. Of course, indexed instructions of a given base address mode can do the same things as their nonindexed counterparts. After all, if the index equals 0, the base address will be unchanged after indexing.

Stacks. Stack data are usually accessed with the 6510's built-in stack addressing mode, which can be considered a mode of indexed addressing. More general stack structures can easily be built and accessed with any of the previous indexed modes except for indexed indirect addressing.

In stack addressing, the index value increases or decreases by a constant value, usually 1, as data are popped off or pushed on the stack, respectively. The main qualitative difference between stack and array addressing is the ebb-and-flow changing of the index found in stack addressing.

FIFOs, linked lists, and hierarchies can also be built using the general indexed addressing modes. Most variations on these structures can easily be implemented by following the accessing principles explained in Chapter 2.

Table 3.2 completes the addressing information begun in Table 3.1. This table summarizes the main indexed addressing modes used to access each type of repeti-

TABLE 3.2 INDEXED ADDRESSING MODES

Structure length	Structure type: Array	Stack	FIFO	Linked list/ hierarchy
< 256 bytes	*Zero-page Absolute Indir index	*6510 stack Absolute Indir index	*Zero-page Absolute	Indir index
> 256 bytes	*Indir index Index indir	*Indir index Index indir	Indir index	Indir index

tion data structure. As before, an asterisk highlights the fastest executing addressing mode for each combination of structure type and length. The word "indexed" has been omitted from most mode titles, except that "indirect indexed" has been abbreviated "indir index," and "indexed indirect" has been abbreviated to "index indir."

Exercise:

Test your retention of the material in this section by answering these questions.

(h) What is indexed addressing?

(i) Describe the absolute indexed addressing process.

(j) Describe the indirect indexed addressing process.

CPU Operations

The remaining half of an instruction, the operation byte, usually determines what is done to addressed data. Its other use is to alter the flow of instructions into the microprocessor for execution.

The mnemonic given this byte describes the action it causes. There are three types of actions controlled by the operations byte, or opcode. The first two affect addressed data. The remaining one affects the instruction flow. These actions are data movement, data transformation, and program structuring.

In this section the opcodes and their mnemonics are grouped and discussed according to the three types of CPU actions. Data transformation operations are further grouped into three functions: arithmetic, shifts, and logic. The mathematics of the latter functions can be reviewed in Chapter 1.

Data movement is the most basic action performed on addressed data, so we will begin with it.

Data movement. 6510 operations always move data to or from a CPU register. There are three possible paths between data source and data destination. They are between register and memory, between register and top of stack, and between register and register. Each of these paths is supported with data movement

operations, usually for both directions of transfer. We can now discuss the data movement operations in terms of the three source/destination paths.

Between Register and Memory. A data transfer from a register to a memory location is called a *store* operation. Stores can be made from the accumulator or either of the index registers to any location in memory.

The operation that moves a byte from the accumulator into a memory location has the mnemonic STA, for STore Accumulator. This operation can be combined with the direct addressing absolute and zero-page modes, and the indexed addressing absolute indexed, zero-page,X, indexed indirect, and indirect indexed modes.

The operation that moves a byte from index register X into memory has the mnemonic STX, for STore X. This operation can be combined with any of the STA direct addressing modes, but only with the zero-page,Y indexed mode.

The operation that moves a byte from Y into memory has the mnemonic STY, for STore Y. STY can be combined with any STA direct addressing mode, but only with the zero-page,X indexed mode.

Store operations have no effect on the flags of the processor status register.

Example:

Write a typical store instruction, combining the operation with an addressing mode.

The instruction for storing the contents of X into the zero-page location specified by zero-page,Y addressing is

```
STX  address,Y     ;zero-page,Y addressing
```

A data transfer from a memory location to a register is called a *load*. Each store operation has a corresponding load operation in the opposite direction. Earlier in this chapter we saw the LDA operation, which LoaDs the Accumulator with a byte from memory. There are also LDX and LDY operations for loading registers X and Y from memory.

The LDA operation adds the immediate mode of direct addressing to the STA addressing modes.

The LDX operation adds the immediate mode of direct addressing and the absolute,X mode of indexed addressing to the STX modes.

The LDY operation adds the immediate mode of direct addressing and absolute,Y mode of indexed addressing to the STY modes.

Load operations affect two flags in the processor status register. Recall that the bits of the status register are set to 1 to indicate or flag different conditions resulting from CPU operations. There is a zero-result or Z flag, a negative-result or N flag, an overflow-result or V flag, a carry-result or C flag, a BCD arithmetic-selected or D flag; an interrupts-enabled or I flag, and a BRK-occurred or B flag. BRKs and interrupts will be described with the program structure operations. The flag bits are placed in the status register as shown in Fig. 3.7.

Bit d5 is unused. The two flags affected by loading a data byte are the Z and N

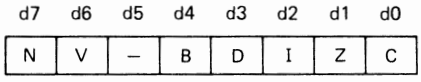


Figure 3.7

flags. Thus if the value 0 is loaded from memory into the accumulator, the Z flag will be set to 1. If a value whose d7 bit equals 1 is loaded into the accumulator, the N flag will be set because the top bit of a 2C number denotes its sign. Even if the data byte is not interpreted as a 2C number by the program, d7 can be used as a Boolean variable and tested with the N flag.

Example:

Write a typical load instruction, combining an operation with an addressing mode.

The instruction for loading a byte into the accumulator from an address specified with indirect indexed addressing is

```
LDA (address),Y
```

Table 3.3 summarizes the most important facts about load and store operations. Corresponding load and store operations are placed together for ease of comparison. Each operation's effect is symbolized within brackets beneath the opcode mnemonic, using the abbreviations M for memory location, A for accumulator, and X and Y for the index registers. The affected flags are also listed.

TABLE 3.3 LOAD AND STORE OPERATIONS

Operation	Direct modes	Indexed modes	Flags
STA [A→M]	Absolute {8D,4} Zero-page {85,3}	Absolute,X {9D,5} Absolute,Y {99,5} Zero-page,X {95,4} Indirect indexed {91,6} Indexed indirect {81,6}	None
LDA [M→A]	Absolute {AD,4} Zero-page {A5,3} Immediate {A9,2}	Absolute,X {BD,4*} Absolute,Y {B9,4*} Zero-page,X {B5,4} Indirect indexed {B1, 5*} Indexed indirect {A1,6}	Z N
STX [X→M]	Absolute {8E,4} Zero-page {86,3}	Zero-page,Y {96,4}	None
LDX [M→X]	Absolute {AE,4} Zero-page {A6,3} Immediate {A2,2}	Absolute,Y {BE,4*} Zero-page,Y {B6,4}	Z N
STY [Y→M]	Absolute {8C,4} Zero-page {84,3}	Zero-page,X {94,4}	None
LDY [M→Y]	Absolute {AC,4} Zero-page {A4,3} Immediate {A0,2}	Absolute,X {BC,4*} Zero-page,X {B4,4}	Z N

A table in this form will follow every operations group we discuss. Included in these tables are the hexadecimal opcode values for each combination of operation and addressing mode, and the instruction's corresponding execution time in microseconds (i.e., one-millionth of a second). These two values are displayed in the form {opcode, time}," following the addressing mode name. The execution times can be used to compute how long a program will take to execute; however, the times listed are based on the assumption that the operations are occurring within a single page of memory (addresses xx00 through xxFF hex). Some instructions straddling a page boundary will take an additional microsecond to execute. Such instructions will be highlighted with an asterisk beside their execution time.

Between Register and Top of Stack. Only the accumulator and status registers can transfer data directly to and from the stack. However, by using accumulator load and store operations, data in memory can be moved into and out of the stack. Finally, using the register-to-register operations of the next section allows data in the remaining registers to be pushed on and popped off the stack via the accumulator.

The ability to save the contents of the processor status register on the stack is extremely important. Sometimes the flow of instructions into the CPU is intentionally but temporarily diverted with a program structuring operation. When this diversion is complete, execution will return to the main stream. If the status of the execution before the diversion is unavailable, the return to the main instruction stream will be of limited usefulness. Saving the status register on the stack before the diversion, and recovering it afterward, solves this problem. This entire process will be described in the program structuring section.

Two operations are used to move data from a register to the top of the stack. They are PHA, for PusH Accumulator byte onto the stack, and PHP, for PusH Processor status byte onto the stack. Neither operation affects the flags, and neither operation has an operand byte since the address is held in the stack pointer.

Two operations are used to move data from the top of the stack to a register. They are PLA, for PulL Accumulator byte from the top of the stack, and PLP, for PulL Processor status byte from the top of the stack. As with the load operations, moving data into the accumulator affects the zero and negative flags. Moving data from the top of the stack into the status register affects all the flags, of course.

In Table 3.4 the effects of push and pull operations are shown with several new symbols. The symbol S represents the contents of the stack pointer, TOS represents the contents of the memory location pointed to by the stack pointer, and P represents the contents of the processor status register.

Between Register and Register. There are two types of register-to-register operations. One serves the general need to move data between registers; these operations are called *transfers.* The other type alters specific bits in the status register; this type consists of the *flag modification* operations.

Transfer operations service the A, X, Y, and S or stack pointer registers. Only three of the possible source/destination pairings are used: A with X, A with Y, and

TABLE 3.4 PUSH AND PULL OPERATIONS

Operation	Direct modes	Indexed modes	Flags
PHA [A→TOS , S+1→S]		Stack {48,3}	None
PLA [S-1→S , TOS→A]		Stack {68,4}	Z N
PHP [P→TOS , S+1→S]		Stack {08,3}	None
PLP [S-1→S , TOS→P]		Stack {28,4}	All

X with S. There are two operations for each of these pairs, one operation for each transfer direction. The mnemonic for each operation consists of the letter T, for Transfer, followed by the letters of the source and destination registers. Of course, all transfer operations use implied addressing.

The operation that moves a data byte from the A register into the X register is called TAX, for Transfer A into X. The operation moving data from X to A is called TXA, for Transfer X into A. The zero and negative flags are affected by both these instructions.

The operation moving data from A into Y is called TAY, for Transfer A into Y. The operation moving data from Y to A is called TYA, for Transfer Y into A. These instructions also affect the zero and negative flags.

The operation moving data from X into S is called TXS, for Transfer X into S. The operation moving data from S into X is called TSX, for Transfer S into X. TXS has no effect on the flags. TSX affects the zero and negative flags. Table 3.5 lists the transfer operations.

Flag modification operations work differently from transfer operations. When

TABLE 3.5 TRANSFER OPERATIONS

Operation	Direct modes	Indexed modes	Flags
TAX [A→X]	Implied {AA,2}		Z N
TXA [X→A]	Implied {8A,2}		Z N
TAY [A→Y]	Implied {A8,2}		Z N
TYA [Y→A]	Implied {98,2}		Z N
TXS [X→S]	Implied {9A,2}		None
TSX [S→X]	Implied {BA,2}		Z N

the opcode for a flag modification or flag mod operation is fetched from the program in memory, it is loaded into the instruction register, or IR, and decoded like any other opcode. Based on this decoding, the microprocessor fixes the value of the appropriate bit in the status register. The process can be thought of as a roundabout data transfer from the IR to the flag register, which is why these operations have been placed in the register-to-register category.

Flag mod operations can place a selected value of 0 or 1 into any of the four following flags: C, I, D, and V.

The first of these, the carry flag, is set to 1 with the SEC or SEt the Carry operation. The C flag is reset by the CLC or CLear Carry operation.

The interrupt flag is set by the SEI or SEt the Interrupt mask operation. The I flag is reset by the CLI or CLear the Interrupt mask operation. Again, interrupts will be explained in the program structure section.

The decimal flag is set by the SED or SEt Decimal mode operation, and reset by the CLD or CLear Decimal mode operation.

Finally, the overflow flag can only be reset, using the CLV or CLear the oVerflow operation.

All flag mod operations imply the status register as the data location; therefore, they all use the implied addressing mode. The flag mod operations are summarized in Table 3.6.

The data movement operations provide the means to move data from anywhere in the computer to anywhere else in the computer. However, the purpose of moving data is usually to support their transformation. We can now discuss the operations that transform data.

Data transformation. In Chapter 1 we explained three ways of transforming binary data: arithmetic, shifts, and logic. There are CPU operations for each of

TABLE 3.6 FLAG MOD OPERATIONS

Operation	Direct modes	Indexed modes	Flags
SEC [1→C]	Implied {38,2}		C
CLC [0→C]	Implied {18,2}		C
SEI (disable ints) [1→I]	Implied {78,2}		I
CLI (enable ints) [0→I]	Implied {58,2}		I
SED [1→D]	Implied {F8,2}		D
CLD [0→D]	Implied {D8,2}		D
CLV [0→V]	Implied {B8,2}		O

these functions. We will explore these operations in the order that we earlier studied the three types of transformations.

Arithmetic. Arithmetic is defined as the four computations of addition, subtraction, multiplication, and division. The 6510, like most microprocessors, has CPU operations only for the addition and subtraction computations. The latter two computations are easily though relatively slowly performed with combinations of other CPU operations. Multiplication and division will be demonstrated after the basic CPU operations become familiar.

There are general- and special-case versions of the addition and subtraction operations. The general case adds or subtracts a byte in the accumulator with a byte in memory. The special case adds or subtracts the value 1 with the value in an index register or in memory. The special-case addition and subtraction operations are known as *incrementing* and *decrementing.*

The 6510 supports arithmetic for two numerical codes: binary and BCD. The microprocessor knows which type of arithmetic to perform by the contents of the decimal flag in the status register. Recall that the flag mod operations SED and CLD are used to set this flag for BCD or binary arithmetic, respectively. Since most programs use just one type of arithmetic, the SED or CLD operation usually appears just once, at the start of the program.

In BCD mode general arithmetic, the microprocessor will compensate for the BCD "no-man's land" of codes from 1010 to 1111. Beware, however; BCD mode increment and decrement operations do not compensate for the illegal code values. It is best to use only general arithmetic operations for BCD arithmetic.

BCD and binary mode arithmetic also differ in their effects on the flags in the status register. In BCD arithmetic the zero and negative flags are fixed according to the result that would occur if the arithmetic were binary. In effect, the CPU does the arithmetic in binary and fixes the Z and N flags before converting the result to BCD. Only the carry flag is fixed consistently with the decimal result. In the program structuring section we will study a group of operations that test individual flags and select an action accordingly. With BCD arithmetic the flag inconsistencies require that these flag-testing instructions be carefully selected. Such instructions should either test only the carry flag, which is accurate in BCD arithmetic, or possibly also test the zero or negative flags, *if* their value is always fixed the same way by the previous flag-affecting operation whether the arithmetic is BCD or binary.

Addition. The general addition operation is called an ADd with Carry, or ADC. The effect of ADC is to add the data byte it addresses with the contents of the accumulator and the value in the carry flag, and place the result in the accumulator. This operation can be written

```
accum + data + carry = > accum
```

If the carry is a 0, the result is just the sum of two addends. If the carry is a 1, the result is one larger.

Multiple-byte additions are simple with the ADC operation. The carry flag addition in the ADC operation allows a carry from the addition of two less-significant bytes to be added automatically into the next-higher byte addition. Of course, the carry must be reset to 0 before the addition of the least-significant bytes in the addition. This is done with an initial CLC flag mod operation.

Example:

Write assembly language instructions to add the numbers 5427h and 1050h.

The addition will be set up as two single-byte additions. Only the ADC operation and a few data movement and flag mod operations are required. The location receiving the least-significant byte of the sum has been symbolically named 'sum'.

```
CLD          ;set up CPU for binary arithmetic
CLC          ;reset the carry flag to 0
LDA  #27     ;load accumulator with LSByte of first addend
ADC  #50     ;add the LSBs, fixing the carry flag
STA  sum     ;store LSB of sum in location 'sum'
LDA  #54     ;load accumulator with MSByte of first addend
ADC  #10     ;add MSBs and carry from the first add
STA  sum+1   ;store MSB of sum in location 'sum+1'
```

Be sure that you fully understand these instructions before going on. Here as elsewhere in assembly language, a potentially imposing group of instructions often proves quite simple when taken one at a time.

The ADC operation can be used with all direct addressing modes except implicit, and all indexed addressing modes except zero-page,Y. ADC fixes the values of the zero, negative, carry, and overflow flags (recall that an overflow occurs when a 2C addition overflows into the d7, or sign, bit).

The special-case addition called the *increment* comes in three forms for the three types of locations it can alter. All forms fix the zero and negative flags.

Memory locations are incremented with the INC operation. It can be combined only with the absolute and zero-page modes of direct addressing, and the absolute,X and zero-page,X modes of indexed addressing.

Register X is incremented with the INX operation. INX is of course addressed only implicitly.

Register Y is incremented with the INY operation, which is also addressed implicitly.

With all three operations, if the data being incremented equal FFh, the sum will wrap around to the value 00.

Example:

Write the assembly language instruction for incrementing an indirect indexed memory location.

Using the symbolic label 'place' for the memory location, this instruction is

```
INC (place),Y
```

In BCD arithmetic, the increment operation is usually replaced by the following actions:

1. A data movement operation to place the target byte in the accumulator
2. A CLC instruction to reset the carry to 0
3. An ADC #1 instruction to increase the target byte by 1
4. A data movement operation to return the altered byte to its source (unless the value of the flags is all that is of interest)

The addition operations are summarized in Table 3.7.

Subtraction. The general subtraction operation is called a subtract with carry, or SBC. SBC subtracts the data byte it addresses and the inversion of the carry flag from the contents of the accumulator and places the result in the accumulator. This operation can be written

accum − data − NOT carry = > accum

In subtraction the inverse of the carry flag acts as a borrow flag. For single-byte subtractions the carry flag can be set so that subtracting NOT carry will have no effect on the result.

In multiple-byte subtractions, if a borrow is required by the subtraction of two less-significant bytes, the carry flag is *reset*. Then the next-higher byte subtraction will, in deducting the inversion of the carry flag, reduce the accumulator-data difference by 1 and complete the borrow. If a subtraction generates no borrow, the carry flag is set, and the next-higher subtraction will deduct the borrow value 0 and leave the difference unchanged.

Because of the borrow flag, the subtraction result can be interpreted either as a

TABLE 3.7 ADDITION OPERATIONS

Operation	Direct modes	Indexed modes	Flags
ADC	Absolute {6D,4}	Absolute,X {7D,4*}	Z
[A+M+C→A]		Absolute,Y {79,4*}	N
	Zero-page,X {65,3}	Zero-page,X {75,4}	C
		Indirect indexed {71,5*}	O
		Indexed indirect {61,6}	
	Immediate {69,2}		
INC	Absolute {EE,6}	Absolute,X {FE,7}	Z
[M+1→M]	Zero-page {E6,5}	Zero-page,X {F6,6}	N
INX	Implicit {E8,2}		Z
[X+1→X]			N
INY	Implicit {C8,2}		Z
[Y+1→Y]			N

positive binary number or as a 2C (possibly negative) number. For instance, the subtraction 30h − 60h produces the value D0h, which is the positive binary result from 130h − 60h (subtraction with borrow) and also the negative 2C result from 30h − 60h (subtraction without borrow). Thus SBC can be used with either straight binary or 2C coded numbers.

SBC fixes the zero, negative, and overflow flags according to the 2C interpretation of the result. Thus d7 is interpreted as the sign bit to set the negative flag.

Multibyte subtractions follow the same pattern as multibyte additions, with the substitution of SBC for ADC and SEC for CLC to set the carry flag and therefore reset the borrow.

Example:

Change the earlier addition problem, 5427h + 1050h, to the subtraction problem 5427h − 1050h.

All that is required is to replace the arithmetic and flag mod operations of the earlier example. Each of these replacements is highlighted with an asterisk. The location receiving the least-significant byte of the result has been renamed 'diff'.

```
    CLD             ;reset flag for binary arithmetic
*   SEC             ;fix the borrow flag for no effect
    LDA  #$27       ;load the accumulator with first LSB
*   SBC  #$50       ;subtract the LSBs, fixing the borrow
    STA  diff       ;store result LSB in location 'diff'
    LDA  #$54       ;load accumulator with first MSB
*   SBC  #$10       ;subtract MSBs and the borrow
    STA  diff+1     ;store MSB of result in 'diff+1'
```

The SBC operation, like ADC, can be combined with all direct addressing modes except implicit, and all indexed data addressing modes except zero-page,Y.

There is another general subtraction operation, called a *compare,* that compares two numbers and determines which is larger, without producing a numerical result. Compare operations differ from the SBC in four ways:

1. A compare discards its numerical result.
2. A compare subtracts from the A, X, or Y registers, whereas SBC always subtracts from the accumulator.
3. A compare subtracts only an addressed data byte and not the inverted carry flag.
4. A compare leaves the overflow flag untouched while fixing all other SBC-affected flags.

The compare operation is useful for testing the contents of a register or an addressed memory location without changing its contents. Compare operations fix the zero, negative, and carry flags.

The compare operation that subtracts addressed data from the accumulator is called CMP. This operation has the same choice of addressing modes as SBC.

The compare operation that subtracts addressed data from X is called CPX. It allows only the direct addressing modes accessing memory (i.e., absolute, zero-page, and immediate).

The compare operation subtracting addressed data from Y is called CPY. Its addressing modes are the same as for CPX.

Example:

Write assembly language instructions to compare the contents of the memory location at address 'loc' with the number 'max':

```
LDX  #max        ;place comparative value in X
CPX  loc         ;compare contents of 'loc' with X
```

The zero flag will be set if the numbers are equal; the carry flag will be set (i.e., the borrow cleared) if 'max' is greater than or equal to the contents of 'loc'. The negative flag will be set if 'max' is less than the contents of 'loc', unless a 2C overflow occurs.

The special-case subtraction called the *decrement* comes in three forms that correspond directly in their addressing modes to the three forms of increments. All decrements fix the zero and negative flags.

Memory locations are decremented with the DEC or DECrement operation. Like INC, it can be combined with the absolute and zero-page modes of direct addressing, and the absolute,X and zero-page,X modes of indexed addressing.

Register X is decremented with the DCX or DeCrement X operation. Like INX, it is addressed implicitly.

Register Y is decremented with the DCY or DeCrement Y operation, which is also addressed implicitly.

The data value 00 decrements to FF hex. As with increments, decrements in BCD mode should be substituted for with data movement and general subtraction operations.

This completes our study of the data transformation operations performing arithmetic. The subtraction operations are represented in Table 3.8. Recall that the overbar symbol means "negate the quantity underneath" (i.e., it is the NOT symbol). The "\" character as a destination means that the operation results are thrown away.

Exercise:

Try the following assignments as a review of the assembly language instructions we have studied thus far.

(k) Write a small sequence of assembly instructions for adding two single-byte BCD numbers.

(l) Write a small sequence of assembly instructions for comparing two BCD numbers.

Shifts. Shift operations move every bit in the accumulator or in a memory location (depending on the addressing mode) one bit position to the left or right. When a byte is shifted, the bit on one end "falls off" and the bit on the other end

TABLE 3.8 SUBTRACTION OPERATIONS

Operation	Direct modes	Indexed modes	Flags
SBC	Absolute {ED,4}	Absolute,X {FD,4*}	Z
[A−M−$\overline{C}$→A]		Absolute,Y {F9,4*}	N
	Zero-page {E5,3}	Zero-page,X {F5,4}	C
		Indirect indexed {F1,5*}	O
		Indexed indirect {E1,6}	
	Immediate {E9,2}		
CMP	Absolute {CD,4}	Absolute,X {DD,4*}	Z
[A−M→ \ ]		Absolute,Y {D9, 4*}	N
	Zero-page {C5,3}	Zero-page,X {D5,4}	C
		Indirect indexed {D1,5*}	
		Indexed indirect {C1,6}	
	Immediate {C9,2}		
CPX	Absolute {EC,4}		Z
[X−M→ \ ]	Zero-page {E4,3}		N
	Immediate {E0,2}		C
CPY	Absolute {CC,4}		Z
[Y−M→ \ ]	Zero-page {C4,3}		N
	Immediate {C0,2}		C
DEC	Absolute {CE,6}	Absolute,X {DE,7}	Z
[M−1→M]	Zero-page {C6,5}	Zero-page,X {D6,6}	N
DEX	Implicit {CA,2}		Z
[X−1→X]			N
DEY	Implicit {88,2}		Z
[Y−1→Y]			N

moves inward and vacates its old end position. The falling-off bit is placed into the carry flag by all shift operations. However, the vacated bit position is filled by shift operations with either a 0, as with two shift operations officially called *shifts,* or with the contents of the previous carry flag, as with two shift operations called *rotates.* We will use the terms "shift" and "rotate" in this more limited sense for the rest of the section.

All shift and rotate operations can be combined with the absolute, zero-page, and implied modes of direct addressing, and the absolute,X and zero-page,X modes of indexed addressing. The implied mode accesses only the accumulator; the symbol A in the operand field indicates the implied mode to the assembler.

The two shift operations are called Logical Shift Right or LSR, and Arithmetic Shift Left or ASL. As you might expect, the first operation moves bits to the "right" or toward lower-bit-position numbers. The second operation moves bits leftward, toward higher-bit-position numbers.

The two rotate operations are called ROtate Right and ROtate Left. They similarly describe the direction of bit movement.

The effects of the shift and rotate operations are illustrated in Fig. 3.8. The name of each assembly language operation accompanies its illustration.

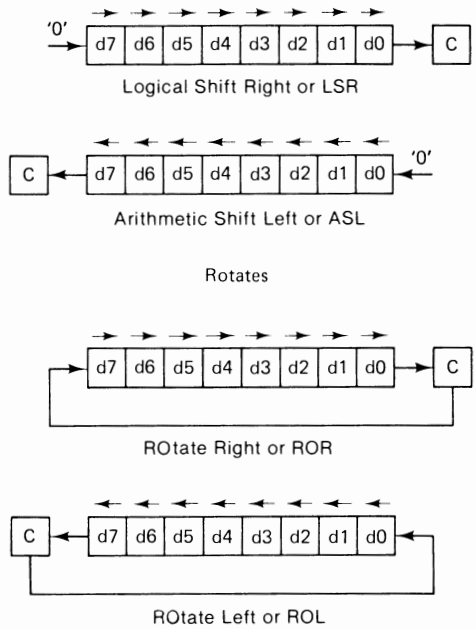


Figure 3.8 Shifts

Shift group operations are used to double or halve a data value, as explained in Chapter 1, or to place a particular bit into the carry flag for testing.

Example:

Write assembly instructions that move the d7 bit of the accumulator into the carry flag for testing and that then restore the accumulator to its initial condition.

```
ASL A       ;place d7 bit of accumulator into carry flag
;(perform the desired test here)
LSR A       ;move previous d7 and other bits back
```

The shift and rotate operations are summarized in Table 3.9. The symbol d(i) represents a bit position from d0 to d7.

Exercise:

Try the following review question

(m) Describe in detail the effect of executing an ROR operation.

Logical Operations. The logical or Boolean functions OR, AND, XOR, and NOT are performed with four CPU operations.

CPU operation ORA, for OR Accumulator, performs a bitwise OR on the contents of the accumulator and an addressed data byte, and places the result in the accumulator. ORA fixes the zero and negative flags.

TABLE 3.9 SHIFT AND ROTATE OPERATIONS

Operation	Direct modes	Indexed modes	Flags
ASL	Absolute {0E,6}	Absolute,X {1E,7}	Z
[d(i)→d(i+1);	Zero-page {06,5}	Zero-page,X {16,6}	N
d7→C, 0→d0]	Implied {0A,2}		C
LSR	Absolute {4E,6}	Absolute,X {5E,7}	Z
[d(i)→d(i−1);	Zero-page {46,5}	Zero-page,X {56,6}	N
0→d7, d0→C]	Implied {4A,2}		C
ROL	Absolute {2E,6}	Absolute,X {3E,7}	Z
[d(i)→d(i+1);	Zero-page {26,5}	Zero-page,X {36,6}	N
d7→C, C→d0]	Implied {2A,2}		C
ROR	Absolute {6E,6}	Absolute,X {7E,7}	Z
[d(i)→d(i−1);	Zero-page {66,5}	Zero-page,X {76,6}	N
C→d7, d0→C]	Implied {6A,2}		C

Operation AND, for AND accumulator, performs the bitwise AND of the accumulator and an addressed data byte, places the result in the accumulator, and also fixes the zero and negative flags.

A variation on AND, called BIT, ANDs the accumulator with the addressed data and throws away the result. BIT also fixes the zero flag in the usual way; however, it fixes the overflow and negative flags in an unintuitive fashion, placing bits d6 and d7 of the memory data byte in the overflow and negative flags, respectively. Since d6 and d7 are moved intact regardless of accumulator contents, BIT can be used to check them without any attention or alterations to the accumulator.

Operation EOR, for Exclusive OR accumulator, performs the bitwise XOR of the accumulator and an addressed data byte, places the result in the accumulator, and fixes the zero and negative flags.

NOT is absent from the 6510 instruction set, but can be simulated with the instruction EOR #%11111111 (recall that Exclusive-ORing a bit with a 1 inverts the bit).

All logical operations except BIT allow any direct addressing mode except implicit, and any indexed mode except zero-page,Y. BIT allows only the absolute and zero-page direct addressing modes.

Example:

Write assembly language instructions that isolate the d1 and d3 bits of memory location 'bitloc' and transport them to the accumulator for further testing.

This task can be performed with two instructions:

```
LDA #%00000101    ;load the accumulator with a mask for zeroing
                  ;all bits but d1 and d3 in the next operation
AND bitloc        ;yields bitloc d1 and d3 bits in accumulator
```

TABLE 3.10 LOGICAL OPERATIONS

Operation	Direct modes	Indexed modes	Flags
ORA [A OR M→A]	Absolute {0D,4} Zero-page {05,3} Immediate {09,2}	Absolute,X {1D,4*} Absolute,Y {19,4*} Zero-page,X {15,4} Indirect indexed {11,5*} Indexed indirect {01,6}	Z N
AND [A AND M →A]	Absolute {2D,4} Zero-page {25,3} Immediate {29,2}	Absolute,X {3D,4*} Absolute,Y {39,4*} Zero-page,X {35,4} Indirect indexed {31,5*} Indexed indirect {21,6}	Z N
BIT [A AND M→Z, d6(M)→V, d7(M)→N]	Absolute {2C,4} Zero-page {24,3}		Z N O
EOR [A XOR M→A]	Absolute {4D,4} Zero-page {45,3} Immediate {49,2}	Absolute,X {5D,4*} Absolute,Y {59,4*} Zero-page,X {55,4} Indirect indexed {51,5*} Indexed indirect {41,6}	Z N

Table 3.10 summarizes the logical operations.

Exercise:

Review the logical operations with the following questions.

(n) How can the Boolean NOT function be performed with a 6510 operation?

(o) How do the BIT and AND operations differ?

Program structuring. The last type of CPU operation is used to give programs their structure. These *program structuring* operations shape a program as it executes, choosing between alternative instructions or reexecuting instructions as needed. The program structuring operations organize the data movement and data transformation operations into a system that does useful work.

Any significant system contains multiple levels of structure. First, there is almost always a central structure that holds the lower levels of structure and the system itself together. Just as the central structure called the skeleton is necessary to support the specialized functions of a higher animal, a central structure is necessary to support and connect the many specialized functions that go into performing a significant task. Skeletons and rugged program structures have many other traits in common. We will use skeletal structure as a familiar angle from which to approach program structure.

A skeleton can be examined at three levels. At the highest level, the skeleton can be considered in the context of the central supporting structure called the spine. Below this level are the skeleton's functional groups, such as the rib cage and the hand. At the lowest level there are individual bones and their connective tissues.

The spine dictates the relationships and general structure of the rest of the skeleton. Looking at an X-ray of the spine alone tells much about an organism's shape, manner of locomotion, and purpose in the ecology.

Looking one level lower at the functional bone groups and their relationships provides a deeper understanding of the day-to-day functioning of the organism.

Finally, studying the lowest level, the individual bones, reveals the smallest details about the organism's structure and internal functioning, information which is usually needed only for special scholarly studies.

There are also three levels of structure in a good program, directly corresponding to the three levels of skeletal structure. From highest to lowest they are the *hierarchy,* the *module,* and the *construct.* They can be visually represented with "X-rays" of program structure, pictorial images that we will learn to draft in Chapter 4.

The hierarchy is the spinal structure of a program. It dictates the relationships and general structure of the major parts of a program. Simply looking over the hierarchy chart reveals the most important facts about a program's shape, manner of execution, and purpose or overall function.

A module is a functional group in the program skeleton. A module contains assembly language instructions which together perform one or more closely related functions, such as moving or transforming a single data structure.

The unity of purpose within a module allows it to be given a short, specific name in the form "imperative verb–optional adjectives–object–optional adverbial phrase." Indeed, with this wording convention it is impossible to give a strong name to a module that does not have a unified purpose. So the naming convention has the double purpose of making the purpose of a module clear to the hierarchy-chart reader and of helping the programmer to create modules of unified purpose. A typical example of a module name might be 'Place Mailing-List Names In Alphabetical Order'.

Hierarchy charts show modules accompanied by the names of the data flows into and out of them. A module can dole out portions of its overall task to smaller modules, controlling the order and circumstances of their execution in the process. The levels of modules this allows combine into a functional hierarchy, hence the name "hierarchy" for the central program structure.

Constructs are the individual bones and their associated connective tissues within the functional groups of the program skeleton. A construct contains a closely related group of assembly language instructions performing a task too small to be considered a module-level function. Constructs are named with the same word format as modules. If the fine detail within a module must be known, its constructs should be examined. Knowledge of this level of detail is necessary during programming and program-correcting or "debugging."

So a hierarchy organizes its component modules to perform a total job. A module organizes constructs to perform a single unified function. A construct organizes individual actions to perform a small task. These three levels of structure can describe actions for performing a task no matter who or what does the performing. Thus they can model a way to perform a house-building task, to be executed by a contractor team, as well as they model a way to perform a data task, to be executed by a computer. If a program performs a task, the three levels of structure are built by surrounding data moving or data transforming operations with program structuring operations. To place the levels of structure in perspective, an example hierarchy chart is shown in the following example.

Example:

Develop a hierarchical structure for calculating monthly loan payments from loan rate, term of payment, cost, and down-payment information.

Figure 3.9 breaks the solution to this problem into three functions and five input/output-related subfunctions. By making some assumptions on the details and order of processing, we could assume that the processing shown in the figure would proceed somewhat as follows.

Ask the user for the dollar cost of the item to be purchased and the amount available for a down payment. Accept these figures and subtract the down payment from the cost to get the principal amount to be loaned. Ask the user for the loan rate in percentage points and for the term over which the loan will be paid off. Accept these figures and use them to calculate the factor in dollars per month per thousand dollars loaned. Multiply the factor and the principal figure to obtain the monthly payment amount. Display the monthly payment.

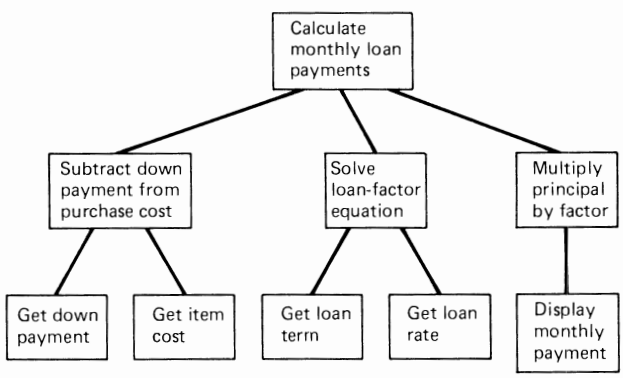


Figure 3.9

Each box in a hierarchy chart represents a module. The lines connecting the boxes indicate higher modules utilizing lower modules to perform their functions. The arrows between the boxes represent data; an arrow into a box represents data used by the module, and an arrow out of a box represents data produced by the module. The name in the top module of the hierarchy chart is the name of the entire hierarchy. Constructs are not shown on a hierarchy chart.

All these aspects of the hierarchy chart will be discussed in Chapter 4, as will methods for creating one as the top level of a program's structure. Creating the lower levels of program structure, modules and constructs, and the assembly operations for doing so, make up the major topic of the rest of this chapter. We will start our expansion with the lowest level of structure, the construct. This will also allow us to continue learning and using assembly language instructions in a relatively simple context. Soon we will see how to combine constructs to form the larger structures called modules.

Constructs. Recall that a construct is a structured group of instructions that performs a single task. Every construct has one entry and one exit point. A construct's *entry point* is a single instruction within it that always executes first. The *exit point* is effectively where all execution paths through a construct come together, before execution passes to the entry point of another construct. The exit point is not always a single instruction in the construct, although it can always be thought of as a single empty instruction executed after the last instruction in each execution path. An *execution path* is a group of instructions that are always executed together within a construct. The concepts of execution paths and exit points from them will become clear when we discuss the different construct types.

Single entry and exit points allow constructs to be built from smaller constructs contained completely within them. This decreases the structural complexity and possibility for errors in the program.

A construct's structure, formed by program structuring operations, imposes an execution pattern on the data processing (data moving and data transforming) operations in the construct. Just as three basic data structures are sufficient to organize data for processing, three basic executing structures are sufficient to organize the data processing operations to access any data. A sequence of data processing instructions accesses a sequence data structure, a repeating group of processing instructions accesses a repetition structure, and a selection of one of several processing-instruction execution paths accesses a selection structure.

The execution structures therefore carry the same names as the data structures: sequence, repetition, and selection. These structures make up the lowest level of structure in a program. As variations on the basic data structures are used to fine-tune a data representation, variations on these basic execution structures are used to fine-tune a construct to the data structure it accesses.

Regardless of internal structure, every construct can be represented in any of three complementary ways. Each way has a special use in program design, although all three are not always used with a given construct. These three representations are structured flowcharts, pseudocode, and templates.

A *structured flowchart* is a picture of processing structure and action. Flowcharts are the only representation we will ever use that describes the processing actions without explicitly considering the data being processed. That omission makes the structured flowchart unsuitable as a solo programming tool, but it can nevertheless be very useful as a precursor to pseudocode.

Structured flowcharts are built with three symbols: the oval, representing the start or stop of construct processing; the rectangle, representing individual processing actions within the structure; and the diamond, representing a choice of execution paths.

Example:

Write a structured flowchart describing a construct that calculates the sales tax on an item purchased in a particular county. The tax rate depends on where it is purchased; if the purchase is made in a city the rate is 6.5%, and if the purchase is made outside a city it is 6%.

The two tax rates indicate that a simple two-path selection of processing action is needed. We show the example construct here in Fig. 3.10 and explain its structure in detail later.

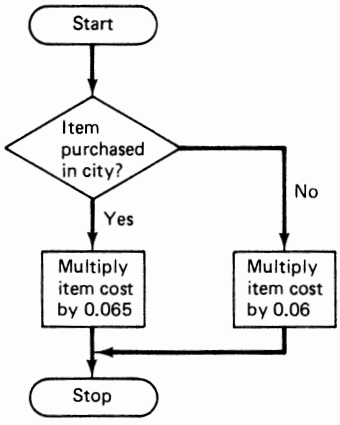


Figure 3.10

Pseudocode is a textual description of a construct's structure and its internal data processing. Each line of pseudocode can describe no more than one processing action or one aspect of the execution structure.

Lines of pseudocode are indented various amounts to show the interrelationships of processing and structural actions. Each processing action is described in pseudocode with an imperative statement like those used in module names (e.g., 'add tax to cost'). Each structuring action is described with a predefined structure statement; these statements and the other details of pseudocode will be revealed as the appropriate construct types are discussed.

Example:

Write pseudocode describing the construct of the preceding example.

The two-path selection construct is called an IF-THEN-ELSE. Its meaning is self-explanatory; its use will be explained with the selection constructs.

```
;calculate sales tax
  IF item is purchased in a city
    THEN set 'sales tax' equal to .065 times item cost
    ELSE set 'sales tax' equal to .06 times item cost
  ENDIF
```

A *template* is the implementation of a construct's structure as a predefined pattern of program structuring assembly language instructions. These patterns correspond to the predefined structure statements of pseudocode. To translate pseudocode into a completed assembly language construct, the programmer selects templates for the structure statements and then fills out the templates with data processing assembly instructions that correspond to the pseudocode data processing lines. Since you have not yet seen the program structuring operations, we will postpone an example of a template until later.

Each type of construct is discussed in its own section below. The program structuring operations that shape each construct, and the usage of the three design tools above, are covered.

Sequences. A sequence accomplishes a processing goal with a series of operations, each executing once in a sequential order. Thus the execution order is always the same within a sequence. Any task that can be done in one pass, where no decisions need to be made as it is done, can be structured as a sequence.

Program structuring operations change the flow of instructions into the CPU; that is, they divert the path of program execution. Since this action is absent in sequences, no program structuring operations are used. Sequences are composed entirely of data movement and data transformation operations.

All three of the design tools can be used with sequences; however, flowcharts are often omitted since the flow of processing actions is easily understood without the picture they provide. A generic structured flowchart for the sequential construct is shown in Fig. 3.11.

The flowchart for a specific sequence construct would substitute a specific task description for 'do task'. The task in the box can often be decomposed into two or more smaller constructs performing the subtasks of the named task. It is interesting that the contained constructs can be nonsequential. As with data structures, constructs can be composed of lower-level structures of differing organization.

A generic pseudocode representation for the sequence construct is the single, trivial line 'do task'. We will see an example of pseudocode for a specific construct shortly.

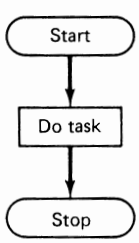


Figure 3.11

The generic template for the sequence construct consists solely of comments accompanying data processing operations, since program structuring operations are not included in sequences. Recall that comments are preceded by semicolons in most assembly languages. The generic sequence template appears as follows:

```
;imperative statement of construct action (e.g., 'add tax to cost')
label       INSTRUCTION 1       ;optional added explanations of
            INSTRUCTION 2       ;processing within the construct
                  •
                  •
                  •
            INSTRUCTION n
;END (construct name)
```

Of course, in a specific template the programmer would supply the imperative statement, label, and comments.

Instruction 1 is the construct entry point and instruction *n* is the exit point. Instructions 1 through *n* process the construct's associated data structure or data elements.

Example:

Use the three design tools together to develop a sequential construct that converts a binary byte into two ASCII bytes representing the hexadecimal values of the binary byte's two nibbles.

This construct might be used in a program that examines and displays the contents of memory locations, for example. We will start with a structured flowchart that shows only the broadest details of the solution (Fig. 3.12).

The next step is to write pseudocode describing the processing shown in the flowchart. First, an imperative statement describing the construct's action must be in-

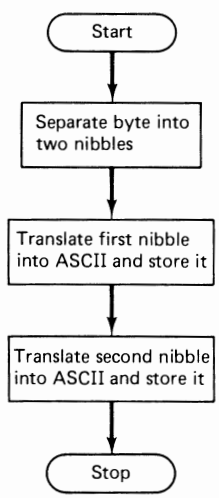


Figure 3.12

serted at the top. We will call the action 'convert a byte into two hex characters', and follow the assembly language convention of preceding it with a semicolon to indicate that it's a comment. The pseudocode is

```
;convert a byte into two hex characters:
    separate byte into two nibbles
    translate the first nibble into ASCII and store it
    translate the second nibble into ASCII and store it
```

Ideally, the next step would be to fill in a template with assembly language instructions. However, each line in the pseudocode above describes a task beyond the capabilities of any assembly language operation. There is, for example, no single instruction that can translate a nibble into ASCII and store it. To obtain pseudocode detailed enough to fill a template from, we must treat each line in our initial pseudocode as a separate task and divide it as before with a flowchart and lower-level pseudocode.

Our flowchart for the first line, 'separate byte into two nibbles', appears as shown in Fig. 3.13. These actions place the upper nibble of the binary byte in the lowest four-bit positions of A, and the lower nibble of the binary byte in the lowest four-bit positions of an index register.

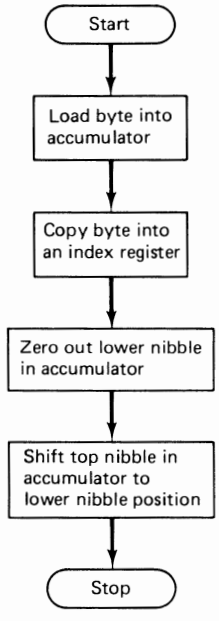


Figure 3.13

The pseudocode for the flowchart in Fig. 3.13 with the index register and shift operations specified is

```
;separate byte into two nibbles:
    load byte into accumulator
    copy byte into X
    zero out lower 4 bits of accumulator
    shift top 4 bits of accumulator four positions right
```

This pseudocode is sufficiently detailed to write assembly language instructions from. The actions required by the second and third lines of the high-level pseudocode are similar, so we will make a general flowchart called 'translate nibble into ASCII and store it' from which to develop the detailed pseudocode. There are several things we know about this function. First, from the expansion of the first line of high-level pseudocode we know that a nibble value is isolated in the lowest 4 bits of the accumulator. We also know that the value of a 4-bit nibble ranges from 0 to F hex. The ASCII codes for characters 0 through 9 are 30h to 39h and the ASCII codes for characters A through F are 41h to 46h. The conversion from nibble value to one of these ASCII values can be made in two major ways: Either the nibble value can be transformed with arithmetic or logic into the ASCII code value, or the nibble value can be used as an index into a table of ASCII code values. In the latter approach the first three bytes in the table would contain the values 30h, 31h, and 32h. We will take the table approach in developing this flowchart and the final assembly language code.

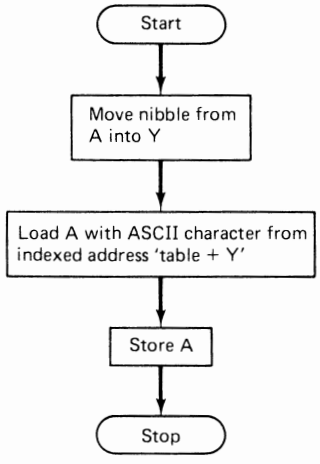


Figure 3.14

The flowchart expanding the task 'translate nibble into ASCII and store it' is shown in Fig. 3.14. We will make two pseudocode translations of this flowchart: one for the byte in the accumulator and one for the byte in the index register. The flowchart fully describes the processing of the accumulator byte, and becomes the following pseudocode:

```
;translate the first nibble into ASCII and store:
    move nibble from A into Y for an index
    load A with ASCII character value at 'table + Y'
    store A
```

To use the pseudocode above with the byte in X, the byte must first be moved into A and its top nibble must be zeroed. With initial lines for these actions, the pseudocode can be used again. The resulting pseudocode expansion is

```
        ;translate the second nibble into ASCII and store:
            move byte from X into A
            zero top nibble in A
            move nibble from A into Y for an index
            load A with ASCII character value at 'table + Y'
            store A
```

Placing the three sections of detailed pseudocode into the initial pseudocode produces detailed pseudocode for the entire construct:

```
;convert a byte into two hex characters:
    ;separate byte into two nibbles:
        load byte in accumulator
        copy byte into X
        zero out lower 4 bits of accumulator
        shift top 4 bits four positions right
    ;translate first nibble into ASCII and store:
        move nibble from A into Y for an index
        load A with ASCII value from 'table + Y'
        store A in memory at address 'high_nibble'
    ;translate second nibble into ASCII and store:
        transfer low-nibble byte from X into A
        zero out top 4 bits of accumulator
        move nibble from A into Y for an index
        load A with table value using Y index
        store A in memory at address 'low_nibble'
END 'convert'
```

Note that the work is done by the lowest-level lines and that the higher-level lines become comments on the processing work done.

The pseudocode is now close enough to assembly language that the template can be filled in. Lines in the detailed pseudocode will become comments in the completed assembly language template. Most lines translate directly into a single assembly language operation. The template will be preceded by an "assignments" section that defines variable names and sets up the table of ASCII characters. The assignment directives =, *=, *=*+, and .BYTE are used. = assigns a numeric value to a symbolic name. *= establishes the memory address at which the following data or program instructions will be placed when the program is first loaded into memory. *=*+, followed by a number, reserves that number of byte locations in memory; if the directive is preceded by a symbolic name, the first address in the reserved area is assigned to that name. .BYTE causes the bytes starting at the program address where this directive is placed to be filled with the byte value or ASCII character sequence that accompanies the directive. The completed template below should make the usage of these directives clear. Comments are preceded by semicolons, as usual.

```
;ASSIGNMENTS
      binary = $5D                  ;assigns value 5Dh to 'binary'
      * = $80                       ;sets first data address to 80 hex
    table                           ;
      .byte '0123456789ABCDEF'      ;stores ASCII table at 80 hex
      lonibl * = * + 1              ;'____nibl' bytes reserved to
      hinibl * = * + 1              ;      store ASCII characters
      * = $2000                     ;address of following instruction
;END ASSIGNMENTS

;convert a byte into 2 hex characters
  ;separate byte into two nibbles
      LDA #binary           ;load number into accumulator
      TAX                   ;copy byte into X
      AND #%11110000        ;zero out lower 4 bits
      LSR                   ;shift accumulator 4 bits right
      LSR
      LSR
      LSR
  ;translate first nibble into ASCII and store
      TAY                   ;move nibble into Y for index
      LDA (table),Y         ;load accumulator with ASCII char
      STA hinibl            ;store character in 'high_nibble'
  ;translate second nibble into ASCII and store
      TXA                   ;transfer low nibble byte into A
      AND #%00001111        ;zero out top 4 bits
      TAY                   ;move nibble into Y for index
      LDA (table),Y         ;load ASCII byte
      STA lonibl            ;store character in 'low_nibble'
;END 'convert a byte'
```

The benefits of using pseudocode and flowcharts together to design code are even more apparent when writing more complex processing structures such as selections or repetitions.

Note that the pseudocode representation provides the comments for the assembly language construct. It is important that you include informative and readable comments with constructs, and pseudocode can provide them. Whether or not the comments are taken from the pseudocode, however, it is very easy to overcomment or to write comments that simply restate the actions of particular instructions. Such comments occasionally serve an educational purpose in this book (as in some of the comments above), but they are in general redundant and distracting. Lines from a pseudocode representation that fit this description should not be included in an actual program.

It is equally important to choose the most descriptive label, address, and data names possible within the character limit. Names and comments are the only direct, written explanations that you will have within a source program (although you will have charts of the structure levels, they are kept separately). Do not be reluctant to

spend a few minutes finding a descriptive name for a problematic label or data structure; the small effort will save you many hours of musings later.

For readability you should also choose one of the letter cases, upper or lower, to notate instructions and structural dividers such as section names (e.g., ASSIGNMENTS), and reserve the other case for variable, label, and address names, and comments as in the preceding example.

There are no variations on the sequence construct.

Exercise:

Use the following questions to review sequential constructs.

(p) How is a sequential structure created?

(q) Under what circumstances should a sequential construct be used?

(r) Design and program a sequential construct that as part of a home-control program, adds together energy-use variables for five home devices and places the result in a 'total energy use' variable. Assume that the energy-consumption values have already been set by another part of the program. Choose your own devices and variable labels.

Selections. Selection constructs have two or more sets of instructions from which one set is chosen and executed by the CPU according to the results of prior instructions. Each set of instructions is called an *execution path.*

The simplest selection construct has just two execution paths. One path must contain at least one instruction, but the other path may contain none. A selection construct having one nonempty execution path and one empty path is called an *IF-THEN* construct, since an execution path is entered only if some condition is satisfied. A selection construct with two nonempty execution paths is called an *IF-THEN-ELSE* construct, indicating the choice of paths depending on whether or not a condition is satisfied. Note that the IF-THEN is a special case of the IF-THEN-ELSE.

A selection construct with three or more nonempty execution paths to choose from can be built with multiple IF-THEN-ELSE constructs, or can be abbreviated as a single construct called a CASE.

All selection constructs are built with one or both of two types of program structuring operations. The first type breaks the flow of execution by diverting the CPU to an address outside the current instruction sequence. This type of operation is called the *JUMP.* Its effect on the CPU was discussed in Chapter 2.

The second type of program structuring operation diverts CPU execution only if a condition is satisfied. This type of operation is called a *BRANCH.* Its ability to choose between execution paths makes program structure possible. The condition a BRANCH checks for is the value in a status register flag. There are eight distinct BRANCH operations, two each for the C, Z, V, and N flags.

The JUMP has two means of addressing the next instruction to execute, corresponding to the absolute and absolute indirect direct addressing modes. The absolute indirect addressing mode is new to us, but it is similar to the zero-page indirect

addressing mode. Both indirect modes use an operand to access a 16-bit address pointer in memory, but absolute indirect does so with a 16-bit address operand, allowing the memory address to reside anywhere in the memory map, while zero-page indirect accesses the memory pointer in page zero with an 8-bit address operand.

In Commodore assembly language format, the JUMP operation has the mnemonic JMP and appears as follows:

```
JMP  addrss      ;absolute addressed, symbolic operand
JMP (addrss)     ;absolute indirect, symbolic operand
```

The BRANCH operation, with mnemonic B followed by a two-letter condition mnemonic, uses a form of indexed addressing. However, instead of using a base address derived from its operand, as in normal indexed addressing, it uses a base address derived by adding 2 to the value of the PC when the BRANCH is fetched. The resulting base address is the address of the instruction after the two-byte BRANCH operation. The BRANCH then adds the 2C operand byte as an index to the base address to produce a final address which is placed in the PC. Instruction execution continues from the new location.

Because the index is two's complement and the initial value from the PC is increased by 2, the final executing address is up to 129 bytes forward of the address of the BRANCH instruction or up to 126 bytes behind it. Thus the BRANCH operand specifies its address *relative* to the address of the BRANCH operation, rather than as an absolute address, and the addressing mode is called the *relative mode* of indexed addressing.

The BRANCHes conditional to the carry flag are called BCS or Branch on Carry Set, which BRANCHes to an out-of-sequence instruction when the carry flag is a 1, and BCC or Branch on Carry Clear, which BRANCHes when the carry is a 0.

The BRANCHes conditional to the zero flag are BEQ of Branch on EQual to zero, which BRANCHes when the zero flag is a 1, and BNE or Branch on Not Equal to zero, which BRANCHes when the zero flag is a 0.

The BRANCHes conditional to the overflow flag are BVS or Branch on oVerflow Set, which BRANCHes when the overflow flag equals 1, and BVC of Branch on oVerflow Clear, which BRANCHes when the overflow flag equals 0.

Finally, the BRANCHes conditional to the negative flag are BMI or Branch on MInus, which BRANCHes when the negative flag equals 1, and BPL or Branch on PLus, which BRANCHes when the negative flag equals 0. The latter condition should more accurately be called Branch on NOT Minus, since a data value of zero will also reset the minus flag.

Table 3.11 summarizes the JUMP and BRANCH operations with their mnemonics and execution effects. The +1 beside each BRANCH execution time means that the BRANCH takes 1 additional microsecond to execute if the condition is satisfied and the BRANCH is taken than if the condition fails. As usual, the asterisk means that the entire instruction takes 1 additional microsecond to execute

TABLE 3.11 JUMP AND BRANCH OPERATIONS

Operation	Direct modes	Indexed modes	Flags
JMP [ADDRESS→PC]	Absolute {4C,3} Absolute-indirect {6C,5}		None
BCS [IF C=1, THEN PC+2+INDX→PC]		Relative {B0,2+1*}	None
BCC [IF C=0, THEN PC+2+INDX→PC]		Relative {90,2+1*}	None
BEQ [IF Z=1, THEN PC+2+INDX→PC]		Relative {F0,2+1*}	None
BNE [IF Z=0, THEN PC+2+INDX→PC]		Relative {D0,2+1*}	None
BVS [IF V=1, THEN PC+2+INDX→PC]		Relative {70,2+1*}	None
BVC [IF V=0, THEN PC+2+INDX→PC]		Relative {50,2+1*}	None
BMI [IF N=1, THEN PC+2+INDX→PC]		Relative {30,2+1*}	None
BPL [IF N=0, THEN PC+2+INDX→PC]		Relative {10,2+1*}	None

if the BRANCH instruction straddles an address page boundary. This time penalty occurs whether or not the BRANCH is taken. INDX refers to the 8-bit operand of the BRANCH instruction. ADDRESS refers to the address value in or pointed to by the operand of the JMP instruction.

To ease the programmer's work, symbolic assembly language format allows the index operand of the BRANCH operation to be written as the symbolic label of the destination address. The assembler converts that symbolic address into the required 2C relative address at assembly time. So, for instance, the instruction using the BRANCH operation BCS to transfer program execution to address 'label' appears in assembly language format as

```
BCS label
```

The address 'label' must, of course, be within -126 to + 129 address locations of the BRANCH operation.

JUMP and BRANCH operations are used together in all selection constructs except the simplest one, the IF-THEN. Since the IF-THEN is a special case of the two-path IF-THEN-ELSE, we start by showing the three design tool representations of the latter construct and will modify the representations to account for the IF-THEN form.

The generic flowchart for the IF-THEN-ELSE construct appears as shown in Fig. 3.15. 'Test' is the program structure selection of the execution path. The IF-THEN flowchart omits one of the 'do task' boxes, and appears as shown in Fig. 3.16.

The generic pseudocode for the IF-THEN-ELSE construct is

```
;imperative statement of construct purpose
  IF condition
    THEN do task 1
    ELSE do task 2
  ENDIF
```

The generic pseudocode for the IF-THEN construct omits the ELSE clause.

Of course, the pseudocode statement IF condition implies "IF condition is true." In assembly language code conditions are flag contents, but it is more useful during design to think of conditions in terms of the overall "barometer" for the upcoming processing. That barometer will be a data relationship or an outcome of previous processes. Later we will see how such conditions can be determined by executing operations and testing the status flags.

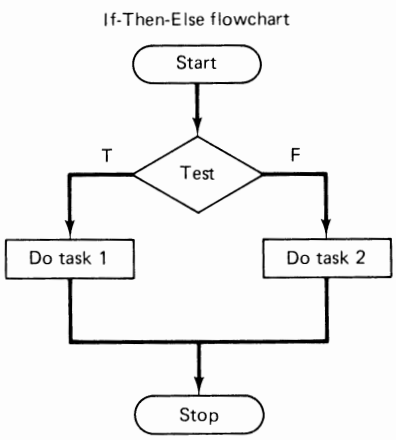


Figure 3.15

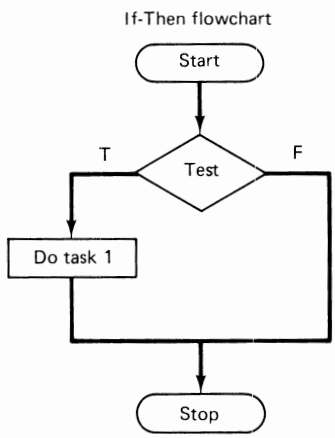


Figure 3.16

Look over the pseudocode for a construct whose choice of execution path depends on the comparative size of two numbers:

```
IF num1 > num2
    THEN do task 1
    ELSE do task 2
ENDIF
```

In this case the processing barometer is found from the relative size of two data elements.

All conditions have Boolean values. 'Num 1 > num 2' is either a true or a false statement. This condition can be translated to a flag value by using a comparison operation with operands Num1 and Num2 to set the value of the negative flag.

It is often very useful to be able to take the logical AND or OR of two or more different Boolean conditions to form a single true or false condition for exiting a construct. This is called a *compound condition*. Compound conditions are often used to save space or to reflect more accurately the barometer on which the processing depends. Also, a single logical barometer will sometimes translate into compound flag tests at the assembly language level.

From Chapter 3 we know that the logical OR of two Boolean values is true if either or both values are true. Thus the condition written 'condition1 OR condition2' is true if at least one of the subconditions is true.

Similarly, the logical AND of two Boolean values is true only if both values are true. Thus the condition written 'condition1 AND condition2' is true only if both subconditions are true.

An IF-THEN-ELSE construct using an OR compound condition is as follows:

```
;imperative statement of construct action
  IF condition1 OR condition2
    THEN do task1
    ELSE do task2
  ENDIF
```

The AND version is similar.

Substituting a compound condition for a simple one changes no other aspect of a construct's generic pseudocode or structured flowchart. The construct's template is affected more, since it shows the assembly language details of the compound condition. With each construct's basic template we will include the AND and OR compound condition variants. Try following the BRANCHes in each template to understand how the conditions are tested.

The general template for the IF-THEN-ELSE construct appears as follows:

```
;imperative statement of construct action
  ;IF condition
  label TEST INSTRUCTION (optional)
        BRANCH (to ELSE clause on NOT condition)
    ;THEN (with imperative statements interspersed below)
        INSTRUCTION 1
        INSTRUCTION 2
              •
              •
              •
        JMP (to entry point of next construct)
    ;ELSE (with imperative statements interspersed below)
    label INSTRUCTION 1
          INSTRUCTION 2
                •
                •
                •
          INSTRUCTION n
;ENDIF (construct name)
```

The IF-THEN generic template omits from the JMP operation at the end of the THEN section down. The BRANCH in the IF section proceeds to the entry point of the next construct.

Replacing the simple 'condition' with the compound 'condition1 AND condition2' in the generic IF-THEN-ELSE template yields

```
;IF condition1 AND condition2
label BRANCH (on NOT condition 1 to ELSE label)
      BRANCH (on NOT condition 2 to ELSE label)
      ;THEN
            INSTRUCTION 1
                  •
                  •
                  •
            JUMP (to beginning of next construct)
      ;ELSE
      label INSTRUCTION 1
                  •
                  •
                  •
;END
```

Note that the THEN section is reached only if condition1 AND condition2 are true.

Finally, replacing the simple 'condition' with the compound 'condition1 OR condition2' in the generic IF-THEN-ELSE template requires an additional label in the THEN section and yields

```
;IF condition1 OR condition2
label  BRANCH (on condition 1 to THEN label)
       BRANCH (on NOT condition 2 to ELSE label)
       ;THEN
       label  INSTRUCTION 1
                    •
                    •
                    •
              JUMP (to beginning of next construct)
       ;ELSE
       label  INSTRUCTION 1
                    •
                    •
                    •
;ENDIF
```

Note that if condition1 is true, the THEN path is BRANCHed to and executed, OR if condition2 is true, the BRANCH to the ELSE path is not taken and the THEN path is executed.

Example:

Write a simple construct that can be used in a space game to keep track of the number of laser hits to your spacecraft and to the spacecraft of your computer opponent, and to keep track of the changing capabilities of the two spacecrafts as a result of the hits they sustain.

Let's assume that the spacecraft that has absorbed the most hits gives up a speed advantage. Giving up an advantage requires comparing numbers of hits and penalizing one spacecraft or the other. This requires the selection of one of two possible actions, which can be represented with a flowchart (Fig. 3.17). We'll call the number of hits to your spacecraft "humanhits", and the number of hits to the computer's spacecraft "computerhits".

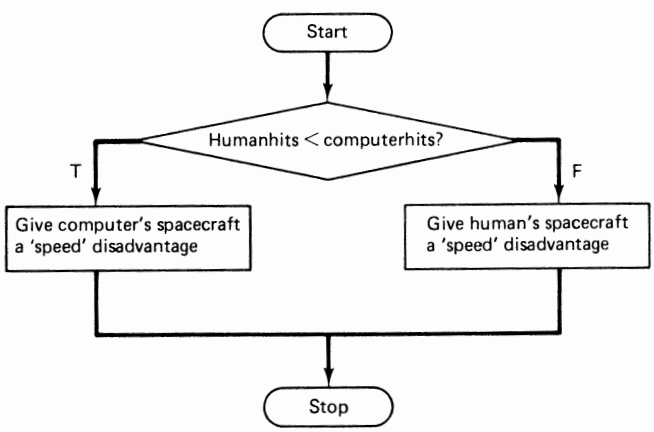


Figure 3.17

In pseudocode, this becomes

```
;place hardest-hit spacecraft at a disadvantage
    IF humanhits < computerhits
       THEN give computer's spacecraft a speed disadvantage
       ELSE give human's spacecraft a speed disadvantage
    ENDIF
```

Note that the ELSE action is only performed if humanhits >= computerhits (i.e., if [NOT (humanhits < computerhits)] is true). This implies that if both sides start out with zero hits, the human is immediately placed at a disadvantage and must fight to gain the advantage. This is typical of many "shoot-em-up" computer games.

Obviously, this pseudocode is not yet ready to be translated into assembly language instructions in a template. The THEN and ELSE lines must each be expanded into multiple low-level lines of pseudocode.

First we must decide on a method for assigning disadvantages. We will do this by defining a 'disadvantage' variable and giving it one of two values, symbolically named 'human' and 'computer' to indicate which spacecraft has taken the most hits. The numeric values of 'human' and 'computer' can be decided upon later. Using this method, the THEN line can be directly expanded into

```
;give computer's spacecraft a speed disadvantage
      Set 'disadvantage' equal to 'computer'
END
```

The ELSE line becomes

```
;give human's spacecraft a speed disadvantage
      Set 'disadvantage' equal to 'human'
```

In this case the final pseudocode is no longer than the initial but has required a design decision to prepare it for coding in assembly language. The final pseudocode representation is

```
;place hardest-hit spacecraft at a disadvantage
    IF humanhits < computerhits
       THEN ;give computer's spacecraft a speed disadvantage
            Set 'disadvantage' equal to 'computer'
       ELSE ;give human's spacecraft a speed disadvantage
            Set 'disadvantage' equal to 'human'
    ENDIF
```

We almost have enough information to fill in a template. All that remains is to decide how the comparison 'humanhits < computerhits' will be made. Recall that numbers are compared by subtracting one from the other. Which number should be subtracted from which?

The subtraction 'humanhits – computerhits' correctly tests the condition 'humanhits < computerhits', since when 'humanhits < computerhits' is True, the sub-

traction result will be negative, setting flags we can test. That is, if 'humanhits' is less than 'computerhits', one of two things will happen: If the size of the result fits in a 2C number (i.e., if 'humanhits − computerhits' is greater than or equal to − 128, the negative flag will be set to show the negative result). However, if 'humanhits − computerhits' does not fit in a 2C number (i.e., if 'humanhits − computerhits' is less than − 128), the result will overflow. The negative flag will be forced to a 0 by the 1-bit in d7, but the overflow flag will equal 1. These aspects of 2C arithmetic can be reviewed in Chapter 1.

Thus a compound flag condition of two possible compound flag conditions after the subtraction 'humanhits − computerhits' is equivalent to the logical condition 'humanhits < computerhits'. The flag condition can be stated as

(N = 1 AND V = 0) OR (N = 0 AND V = 1)

As we said earlier, testing a single processing barometer sometimes requires testing more than one flag.

We look over the IF-THEN-ELSE templates for both OR and AND compound conditions, and build a template based on both. Since the rest of the pseudocode is ready to be converted into assembly language, we will fill in the template. The ASSIGNMENTS section is included to show how the variables and their allowable values can be defined. The variables 'humanhits' and 'computerhits' are assumed to exist elsewhere in the game program and to already contain values.

```
;ASSIGNMENTS
      human = $FF          ;constant definitions
      comptr = $00         ;(labels are limited to six characters)
      * = $80              ;'humanhits' will be at address 80h
      huhits * = * + 1     ;reserves 1 byte for each variable
      cmhits * = * + 1
      disadv * = * + 1
;END ASSIGNMENTS

;place hardest-hit spacecraft at a disadvantage
;IF humanhits < computerhits:
  ;that is, if ((N = 1 AND V = 0) OR (N = 0 AND V = 1))
  ;after subtraction 'huhits − cmhits'
 if       SEC              ;compare 'huhits' and 'cmhits'
          LDA huhits
          SBC cmhits
  c1      BPL c2           ;test first half of condition:
          BVC then         ;(N = 1 AND V = 0)
  c2      BMI else         ;test second half of condition:
          BVC else         ;OR (N = 0 AND V = 1)
  ;THEN ;give computer's spacecraft a speed disadvantage
  then    LDA #comptr      ;set 'disadvantage' = 'computer'
          STA disadv
  ;ELSE ;give human's spacecraft a speed disadvantage
   else   LDA #human       ;set 'disadvantage' = 'human'
          STA disadv
;ENDIF 'disadvantage'
```

A single IF-THEN or IF-THEN-ELSE construct always contains a Boolean or two-valued condition. If a condition has three or more possible values with each having its own distinct processing implications, IF-THEN-ELSE constructs can be combined to select the corresponding execution paths.

For instance, most central air-conditioning systems perform three distinct actions: cooling, heating, and waiting. The temperature can be thought of as a three-valued variable that always implies one of these actions. Each of the three values, then, is a temperature range. The thermostat will check the temperature and select the execution path that initiates either cooling, heating, or waiting. If the thermostat is computer controlled, as many are, the selection will be made by some sort of selection structure in a program.

Selections based on three or more possible values can be built from combinations of IF-THEN-ELSE constructs with Boolean conditions for each value. For instance, the computer thermostat can have one IF-THEN-ELSE for the Boolean condition 'temperature = too cold' TRUE or FALSE, and a following IF-THEN-ELSE testing the condition 'temperature = just right' TRUE or FALSE. If the latter condition is FALSE, the only remaining possibility is 'condition = too hot'.

Thus a non-Boolean condition having n possible values can be divided into "$n - 1$" Boolean conditions individually testing for value 1, value 2, and so on, through value $n - 1$. This process can more easily be understood by seeing how the IF-THEN-ELSE constructs are combined.

The first IF-THEN-ELSE tests for the first possible value of the condition variable. Its THEN section contains the execution path for the first value, and its ELSE section contains an IF-THEN-ELSE that tests for the second possible value of the condition variable. In a similar way, the second IF-THEN-ELSE selects between the second possible value and all remaining values. This pattern continues through the IF-THEN-ELSE that tests for the $(n - 1)$th possible value. The latter construct selects between the execution paths for the $(n - 1)$th and nth condition values.

Example:

Show generic pseudocode for a construct that selects between four execution paths depending on the contents of a four-valued variable.

The pseudocode is

```
;select between four processing options
   IF variable has value 1
      THEN do task 1
      ELSE IF variable has value 2
               THEN do task 2
               ELSE IF variable has value 3
                        THEN do task 3
                        ELSE do task 4
                    ENDIF
           ENDIF
    ENDIF
```

Combining IF-THEN-ELSE constructs in this way is called *nesting*. As you can see, with just a few levels of nesting the structure of the overall construct is obscured. Because of this, it is usually best not to exceed the three levels of nesting shown in the example above.

This limitation makes nested IFs inadequate for the general case of selecting between multiple processing options with a multivalued variable. A much clearer representation of multiple selection is given by the CASE construct. We will define this construct with the three design tools.

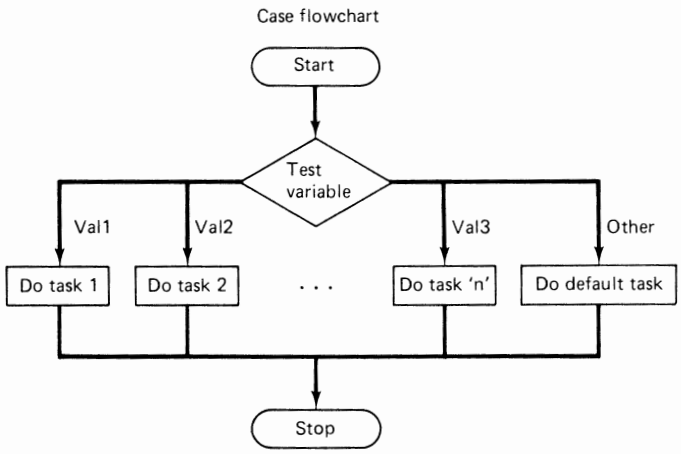


Figure 3.18

The generic flowchart for the CASE construct is shown in Fig. 3.18. The default task is performed only if the variable has a value other than that of the individually defined values 1 through *n*. The default task might react to all erroneous values of the variable in the same way, for instance.

The generic pseudocode for the CASE construct is

```
;imperative statement of construct action
;CASE   variable of
            value 1:    do task 1
            value 2:    do task 2
            value 3:    do task 3
               •
               •
               •
            value n:    do task n
            other:          do default task
ENDCASE
```

Finally, the generic CASE template appears as follows:

```
;imperative statement of construct action
;CASE variable of
        ;value 1
            TEST INSTRUCTIONS
            BRANCH on (NOT value 1) to value 2 case
            PROCESSING
            JUMP (to entry point of next construct)
        ;value 2
            TEST INSTRUCTIONS
            BRANCH on (NOT value 2) to value 3 case
            PROCESSING
            JUMP (to entry point of next construct)
        ;value 3
            •
            •
            •
        ;value n
            TEST INSTRUCTIONS
            BRANCH on (NOT value n) to 'default' case
            PROCESSING
            JUMP (to entry point of next construct)
        ;'default'
            PROCESSING
;ENDCASE (construct name)
```

Compound conditions can also be used. They are built the same way as in IF-THEN-ELSE templates.

The 'value' sections of the CASE template are so similar to the THEN section of an IF-THEN-ELSE template that we will omit a CASE example and go on to the final type of construct, the repetition.

Exercise:

Answer the following questions to review selection constructs.

(s) Describe how the JUMP operation is used in an IF-THEN-ELSE template. Do the same with the BRANCH.

(t) When would a CASE statement be used instead of an IF-THEN-ELSE?

(u) Use all three design tools to write a construct that controls an air-conditioning system. Temperature will be the condition for selecting an action. Assume that 'temp' is a one-byte variable given its value elsewhere in a larger program. Use its value to set another one-byte variable called 'cycle' to one of three values: 'heating', 'waiting', or 'cooling'. Heating should be commanded for 'temp < 68', waiting for '68 < = temp < 78', and cooling for 'temp > = 78'. These ranges will require the use of compound conditions. Try writing this construct first as nested IF-THEN-ELSE and then as a CASE construct.

Repetitions. A repetition construct performs a single task by repeatedly executing a single group of instructions until a test, which can be located anywhere

among the instructions, is satisfied. Execution then diverts to the entry point of another construct. Because of this circular execution pattern, the repetition construct is also called the loop.

The choice between continuing or exiting the loop is made using a BRANCH instruction. If the test condition is compound, a combination of BRANCH and JUMP instructions may follow the test as in IF-THEN-ELSE constructs. A JUMP instruction is located at the physical end of the loop to return execution to the physical beginning of the loop.

By allowing the test to reside anywhere within the repetition structure, a single byte of repetition construct serves all programming needs. The full name of this general repetition construct is the LOOP-EXITIF-ENDLOOP. Its three representations follow.

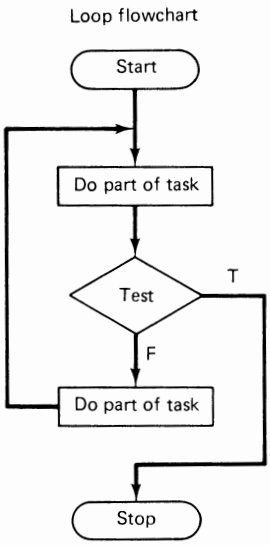


Figure 3.19

In generic flowchart form the LOOP construct appears as shown in Fig. 3.19. Either of the 'do part of task' blocks can be omitted to allow the test to be placed at the beginning or end of the construct.

The generic pseudocode for the LOOP construct is

```
;imperative statement of construct action
LOOP
        do part of task (optional)
        EXITIF condition
        do part of task (optional)
ENDLOOP
```

Finally, the generic template for the LOOP construct is

```
;imperative statement of construct action
;LOOP
label:
        ;do part of task (optional)
   ;EXITIF condition
    BRANCH (on condition to entry point of next construct)
        ;do part of task (optional)
        JUMP label
;ENDLOOP (construct name)
```

Compound conditions for the EXITIF test are treated as they are in IF-THEN-ELSE constructs.

LOOPs usually step through a repetitive data structure using an index in X or Y. If the index equals the number of bytes in the array, it can be decremented to work backward through the array and to produce an easily detected status condition (e.g., 0 or minus) when processing is finished. Although processing backward seems odd at first, it avoids the time-consuming strategy of incrementing an index from 0 and comparing to the number-of-bytes figure on each pass through the LOOP.

An interesting variation of the LOOP construct omits the EXITIF test to produce an *infinite loop*. Once started, an infinite loop cannot be stopped except by turning off or resetting the computer, or by an outside intervention called an interrupt. We will study interrupts later in this chapter. Infinite loops are sometimes useful for such tasks as monitoring and reacting to a condition that occurs repeatedly but at unpredictable intervals. However, the unstoppable execution of an infinite loop limits its usefulness to relatively few situations.

We are now ready for an example of a typical LOOP construct.

Example:

Some tasks require that data in a contiguous memory area be checked to ensure they have not somehow been changed. For instance, data sent to the computer by a peripheral should be checked after their reception, and data written into memory as part of a memory test should also be checked for changes. One way of checking data is to accompany the original data with a data element containing the lowest *n* bits of the sum of all the data bytes, with *n* most commonly equaling 8 or 16. This accompanying data element is called a *checksum*. Creating or verifying a checksum requires that the data in memory be treated as a repetition of bytes, and which therefore calls for the use of a repetition construct.

Assume that an array of bytes has somehow been placed into memory. The byte preceding the array is located at address 'numbyt' and has as its value the total number of bytes in the array plus 2, to account for itself and for a byte following the array. The byte following the array has the 8-bit checksum from adding the array bytes with 'numbyt'. Use the design tools to write a construct that verifies the checksum for the data in memory.

This is a simple enough task that writing a flowchart would not be of much help to us. The repetition construct will be contained in a sequence that sets up, performs, and reacts to the checksum calculation. We will begin with pseudocode.

```
;calculate the checksum of an array in memory
  read 'numbyt' to determine the length of the data area
  add all bytes in the data area except for the checksum
      into a one-byte location
  compare the sum with the checksum byte in the data area
  set a warning flag in memory if the sum <> checksum byte
```

A repetition construct will be needed to perform the 'add numbyt . . .' line. To allow the numbyt value 1 to represent the first byte in the data area, the base address for indexing through the array will be set to one address below the beginning of the array. This is done in the pseudocode expansion below. Indexing will proceed from the end of the array to the beginning, as explained earlier. Since the first line and last two lines are almost at a low-enough level to code, we will expand all three lines at once. This produces

```
;calculate the checksum of an array in memory
    ;read 'numbyt' to determine the length of the data area
      load X with the contents of 'numbyt'
    ;add the bytes in the data area, except for the checksum,
    ;into a one-byte location
      subtract 1 from X (to point below the checksum)
      set A equal to 0 (use A as the one-byte location)
      LOOP
            add byte at address 'numbyt'-1+X to A
            subtract 1 from X
        EXITIF X equals 0
      ENDLOOP
    ;compare sum with the checksum byte following the array
      load X with the contents of 'numbyt'
      compare A to the checksum in location 'numbyt'-1+X
    ;set a warning flag in memory if sum <> checksum byte
      IF A <> contents of checksum byte
        THEN set contents of location 'data' equal to 'nvalid'
        ELSE set contents of location 'data' equal to 'valid'
      ENDIF
END (calculate)
```

The completed template below will omit an ASSIGNMENTS section and any comments from the detailed pseudocode which translate directly into assembly instructions. The result is

```
;calculate the checksum of an array in memory
    ;read 'numbyt' to determine the length of the data area
        LDX numbyt
    ;add the bytes in the data area, except for the checksum,
    ;into a one-byte location
        DEX          ;subtract 1 from X to point below the checksum
        LDA #$00     ;use A as the one-byte receiving location
```

```
;LOOP
 loop
       ADD numbyt-1,X
       DEX
     ;EXITIF X equals 0
      BEQ compar
;ENDLOOP
 JMP loop
;compare sum with the checksum byte following the array
 compar
     LDX numbyt
     CMP numbyt,X      ;compare A to the checksum
;set a warning flag in memory if A <> checksum byte
  ;IF A <> contents of checksum byte
       BEQ else
     ;THEN set contents of location 'data' = 'nvalid'
       LDA #nvalid
       STA data
       JMP done
     ;ELSE set contents of location 'data' = 'valid'
      ELSE
       LDA #valid
       STA data
  ;ENDIF
;RETURN
;END (calculate)
```

Exercise:

Answer the following question to review repetition constructs.

(v) Use all three design tools to write a repetition construct that checks for a time-of-day variable, sets a 'sprinkler system' variable to the value 'ON' at 9:00 A.M., and sets the 'sprinkler system' variable to the value 'OFF' at 9:20 A.M.

Modules. Recall that a module is a group of one or more constructs that perform a complete function in a program. Thus modules are internally similar to constructs and can be implemented by using the same three design tools.

The feature that distinguishes modules from constructs is their scope. While constructs often organize instructions to perform small actions that are by themselves unmeaningful to any overall task, modules always perform major actions that are critical functions of an overall task.

An analogy for this distinction is found in the typical auto repair manual. Most such manuals describe how to repair the major systems of a car in a step-by-step fashion. If you close your eyes, open the manual to a page at random, and read any one sentence in a repair description, you probably will not learn anything about what it takes to repair that automobile system, and you may not even understand what the sentence means because it is out of its context. However, if the repair in-

structions are grouped by major activities such as "remove the carburetor," "disassemble the carburetor," and "reassemble the carburetor," and if the sentence you read is one of these section headings, you will immediately learn one of the major steps in the overall task of rebuilding a carburetor and will likely understand at least its top-level meaning. The major activity sections in a repair sequence correspond to modules in a program, while the individual lines or groups of lines within a section correspond to individual constructs in a module.

The 6510 provides a means for executing a module as a separate entity that does its function and disappears without a trace. You could think of a module as an expert who is called in to do a job that is part of an overall project. The expert does his or her job and then leaves, leaving you mystified about how the job was done but ready to go on to the next job in the project.

A program then becomes a collection of independent functions organized as modules that are called in as needed by other modules, according to the order shown in a module hierarchy chart. Each module specializes in a single function, which is used by the program without a need to know how the function is performed. Insulating each module from all others minimizes the need to pass data between them, which in turn minimizes program complexity and programming difficulty.

Each module is in effect a small system that takes available data and invisibly turns it into desired output data. A system with these characteristics is called a *black box.* A black-box system is also designed to yield helpful and nondestructive results when given the wrong data; it monitors the data it is given and then takes appropriate action depending on whether the data are valid or invalid.

Thus a structure chart such as "CALCULATE MONTHLY LOAN PAYMENTS" shows modules as black boxes with data inputs and outputs, but without discernible inner workings. The structure chart is a picture of the hierarchy and the modules it controls. The modules that are physically higher on the page in a structure chart have more authority; that is, they call for the services of the modules connected beneath them.

Example:

In Fig. 3.20, A calls for B and C, B calls for D and E, and C calls for F.

The instructions in a module reference only the modules on the next-lower level of the hierarchy. This separates the overall task into more manageable pieces. This referencing is done with the CPU capability mentioned earlier in this section. Within the program structure group of operations are a set of operations that allow one module to call for the services of another. This group includes operations that divert program flow from a calling module to a called-for module, and operations that return program flow from the called module to the point in the calling module from which the program flow departed.

The operations that modules use to call for other modules are known as *CALL* operations. The standard CALL operation on the 6510 is named JSR, for Jump to SubRoutine.

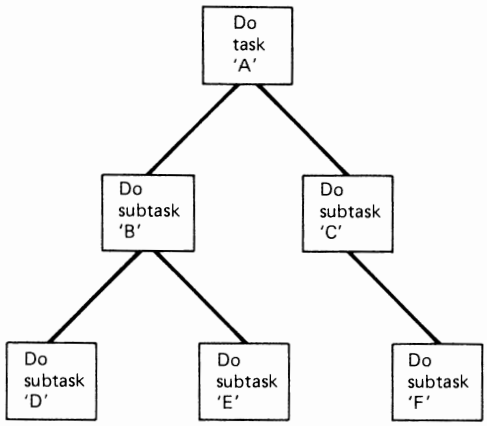


Figure 3.20

JSR is a byte-long operation that is always followed by a two-byte-long absolute address. It adds 2 to the address stored in the PC and pushes the resulting number on the stack (PC + 2 is the address of the second byte of the JSR address operand). It then places its own absolute address operand into the PC, transferring program control to the new address. In our module-based type of programming, the JSR address operand is the entry point of a module, and hence the entry point of the first construct within that module.

The exit point operation of a called module returns execution to the calling module, and is therefore called a *RETURN*. The RETURN operation that corresponds to the JSR is called an RTS or ReTurn from Subroutine. This operation removes the top two bytes on the stack, which will be the address deposited by the last JSR unless the program has altered the stack, and places it in the PC. The PC is then incremented to point to the instruction after the original JSR, and execution continues from there.

When a module calls a module that calls a module and so on, the return addresses are placed on the stack in the calling order. These modules will return in reverse order to their calls; which requires that the return addresses be available in reverse order to their storage. This requires a LIFO structure, which is conveniently available in the 6510 stack.

Example:

Show a simple example of a module calling another module.

The following assembly code shows the use of the JSR and RTS instructions to implement two module levels.

```
;wait during time period equaling 'count' times delay-module
 wait
     ;LOOP
      loopwt
```

```
            ;EXITIF count-down completed
             DEC count
             BEQ timeup
                 ;delay another time period
                      JSR delay
  ;ENDLOOP
   JMP loopwt
  timeup
       RTS          ;RETURN to calling module
;END (wait)

;delay a set time period
 delay
     LDA #$FF
    ;LOOP
     loopdl
      ;EXITIF time period is up
       DEC A
       BEQ done
    ;ENDLOOP
     JMP loopdl
    done
      RTS          ;RETURN to module 'wait'
;END (delay)
```

We will see more significant applications of modules calling other modules in Chapter 4.

A module can call itself instead of another module; this is the programming application of the concept called *recursion*. If you use recursion, you must be careful to preserve any flag, register, and other data used by the module so that they can be recovered after the return. A more detailed discussion of recursion is beyond the scope of this book.

The 6510 has an input called IRQ. Sending a 1 (5-volt) pulse on this line has a similar effect to executing a call; the line allows a chip or other external device to call for the execution of a section of program code in much the same way as a module can call another module with a JSR. This hardware call is known as an *interrupt,* from which the line's name IRQ, or Interrupt ReQuest, is derived.

We cannot execute an IRQ as a program operation, but there is a system clock chip that we can program to generate interrupts periodically. This clock will be discussed further in Chapter 5.

For the 6510 to react to an IRQ pulse, interrupts must first be enabled by preparing the 6510 to accept an interrupt. Executing the CLI or CLear Interrupt disable flag mod operation will accomplish this. Then, reception of a pulse on IRQ causes the 6510 to first set the interrupt disable flag so that succeeding interrupts will not break into the processing requested by the current interrupt. The 6510 then

pushes the most significant byte, or MSByte, of the current PC address onto the stack, followed by the PC least significant byte, or LSB, and the contents of the flag register. It then fetches an indirect address from memory locations FFFEh and FFFFh and places it in the PC. The contents of these bytes depend on the current configuration of the Commodore 64, a topic that is discussed in Chapter 5. The PC then begins executing an *interrupt-handling routine,* which investigates and responds to the interrupt condition and which would be written as a module. Toward the end of the interrupt handler the interrupts must be reenabled with a CLI instruction. Although interrupts are disabled automatically at the beginning of the interrupt-handling routine, it is often useful to disable them manually. This is done with the SEI or SEt Interrupt disable flag mod operation.

The exit point for the interrupt-handling routine is an instruction called RTI, for ReTurn from Interrupt. This instruction places the flag byte from the stack into the status register and places the return address from the stack into the PC.

There is an instruction that simulates the effect of an interrupt request on IRQ. It is called a BRK, for BReaK, and it causes the 6510 to perform the same actions as an IRQ pulse, except that 'PC + 2' is pushed on the stack instead of 'PC', and the B flag is set to 1. As with IRQ-called routines, RTI rather than RTS is used to return from a routine called by a BRK.

Although the return address is PC + 2, BRK is a one-byte instruction. The return address on the stack will have to be popped off the stack and decremented if the BRK is used as a permanent part of a program. BRK, however, is usually used for debugging an existing program, where it is placed over a program instruction. The debugging program that places the BRK has to determine the length of the overwritten instruction, and, if it is other than two bytes, to adjust the return address on the stack during execution of the BRK routine.

The CALL and BRK operations are summarized in Table 3.12. TOS represents "Top Of Stack." (FFFE, FFFF) means "the contents of locations FFFE and FFFF."

With these operations a black-box module or a black-box peripheral device can activate another black-box module. This is only part of the job, however, since data must also usually be communicated between the black boxes. This communication is called *parameter passing.* Parameters can be passed between black boxes by having the activating box place the data in one or more registers, memory locations, or stack locations before the lower box is activated. Large amounts of data can be passed from various areas of memory by passing the data address instead of the data itself. Of course, all the black boxes must agree where the data or pointers will be placed. Parameter passing will be demonstrated in the examples of Chapter 4.

You are now familiar with every operation the 6510 performs except one. That instruction belongs with none of the groups we have discussed, for it does nothing but consume time and space. It is therefore called the NOP or No OPeration. It can be used to pad a loop so that it takes a specific execution time, to create a loop whose only purpose is to pause or delay, or to reserve space for the possible future instruc-

TABLE 3.12 CALL AND BRK OPERATIONS

Operation	Direct modes	Indexed modes	Flags
JSR [PC+2→TOS; OPERAND→PC]	Absolute {20,6}		None
RTS [TOS→PCL, S+1→S, TOS→PCH, S+1→S, PC+1→PC]	Implied {60,6}		None
BRK [PC+2→TOS, S+1→S, (FFFE,FFFF)→PC]	Implied {00,7}		B I
RTI [TOS→P, S+1→S, TOS→PCL, S+1→S, TOS→PCH S+1→S]	Implied {40,6}		All

tions that might be inserted at a point in the program. The NOP can be summarized as follows:

Operation	Direct modes	Indexed modes	Flags
NOP	Implied {EA,2}		None

Exercise:

Review the module level of program structure with the following questions.

(w) Explain the difference between the module and basic construct levels of program structure.

(x) Explain the difference between the JSR and IRQ methods of calling a module and returning from it.

Hierarchies. A hierarchy defines the central order in a program by its pattern of module calls and parameter passing. The calling structure is created with CALL and RETURN operations. The parameter-passing structure is created with data movement operations. The organization is like that of a hierarchical data structure, with modules whose functions have family relationships being grouped under the same branches.

The challenge in creating hierarchies is in exploiting the family relationships between module functions to produce the most efficient program possible. This subject is too large to cover in this chapter; it and the other issues of program planning are discussed in Chapter 4.

Thus we have reached the point where we have to consider how programs are planned. Although planning the overall structure of a program is beyond our means at this point, planning the internal workings of its modules and constructs is not. A module or construct task is of more limited scope than a program task, yet even these limited tasks often allow for more than one solution method. For any given module or construct task, some methods will be faster or consume less memory space than others. Selecting the most efficient method of performing a task and implementing it in the most efficient manner possible is the programming practice called *optimization*.

OPTIMIZATION

As our definition indicates, optimization is a two-stage process. The first stage of optimization is to select the most efficient method for performing a task. This requires both identifying candidate methods and evaluating their relative efficiencies. A candidate method is a method that performs a task in a finite number of steps, where each step is clearly defined and leads without ambiguity to another step. Such methods are called *algorithms*. The optimization process emphasizes this stage, since the choice of an efficient algorithm can produce exponential savings in time or memory space.

The second stage of optimization is to implement the selected algorithm in the most efficient manner possible. This stage exploits the machine characteristics of the CPU to make the most efficient use of the instruction set, registers, and memory map. The gains available at this stage range up to another 25 percent speed or space improvement. This type of optimization was applied to the construct examples in the choice of data placement in registers and memory.

These two stages are discussed in the following two sections.

Algorithm Optimization

With any module or construct the first use of a design tool, be it the flowchart or pseudocode, represents the algorithm for performing that structure's task. Thus the first step in writing a module or construct is to select an algorithm for the task to be performed.

Algorithms have already been devised for most of the major tasks that a computer performs. *Beyond Power Programming,* the advanced adjunct to this book, lists many of them in the pseudocode format used throughout this text. Professional programmers often use algorithm references, although their representation form is usually more cryptic than the style of pseudocode used herein.

Normally, more than one algorithm is available to perform a given task. The appropriate choice depends on the type of efficiency being sought. Either execution time or memory space can be minimized by choosing the proper algorithm. According to the principle called the *time-space trade-off,* efficiency in one criterion is obtained at the expense of the other. So a task often will have preferred algorithms for speed, for brevity, and for the best compromise between the two. Speed is usually considered to be the more important efficiency criterion, since programs often leave memory space available that can be traded for shorter execution time. In any case, choosing the most optimal algorithms will produce the fastest-executing program for the memory, and possibly disk, space available.

An example of the time-space trade-off is found in the binary-to-ASCII conversion in the sequential construct section. This construct obtained its ASCII values from a look-up table. The look-up table algorithm was used for two reasons. First, it is the fastest algorithm for binary-to-ASCII conversion. Second, it is the only major conversion algorithm that can be placed in a sequential construct, and the other types of constructs had not yet been discussed.

If we had considered algorithms of other processing structures, a more space-efficient algorithm would have been available. It is the computational algorithm mentioned at the beginning of the conversion example. The computational algorithm computes the ASCII values representing a binary byte by choosing one of two possible values to add to each nibble in a binary byte. Which value is added depends on whether the hex digit for that nibble will be represented by a number or a letter. The choice of the value to add is a selection process, requiring a selection structure. The computational algorithm is at once slower and shorter than the look-up-table algorithm.

The look-up and computational algorithms clearly demonstrate the time-space trade-off. Similar algorithms of both these types are also used in other types of conversion tasks. For instance, obtaining sines and cosines for given angles can be done with a look-up table or by computation. The look-up-table algorithm for sine calculation quickly lifts a needed sine from a large table of sine values. The computational algorithm performs a complex and time-consuming but relatively space-thrifty calculation of a sine value. Contrasting the ASCII and trigonometric tasks suggests, correctly, that the greater the complexity of the task being performed, the greater the time-space trade-off between its various candidate algorithms will be.

Under certain circumstances a look-up-table algorithm can replace a program CASE structure and save both time and space. This is possible when the CASE execution path options all contain the same type of data movement operations. Thus if the options of a particular CASE construct assign different values to a single variable, the CASE selection variable can often be converted into an index for a look-up table of the different option values. One of these values will be retrieved during execution and placed in the target variable. This strategy can also be helpful when one or two CASE options include additional operations, if the look-up-table algorithm is followed by simpler IF-THEN or CASE constructs than the original CASE. This trade-off will have to be examined on a case-by-case basis.

Whether candidate algorithms are obtained from a reference manual or from programmer experience and creativity, making a reasonable choice between them requires some standard of comparison. The standard we will use determines the relative time efficiency of different algorithms with differing amounts of data. Accounting for differing amounts of data is important because a task in a program may handle different size data structures at different times. An algorithm that works quickly with small data structures but that slows disproportionately as data are added must be exposed by any acceptable standard. Of course, such an algorithm can still be an appropriate choice if the task it performs will never process more than a few data elements or if no better algorithms have been devised.

Our standard is called the *order* of an algorithm. It expresses the ratio change p in the number of data processing (transformation and movement) operations required to complete a task as a simple function of the ratio change n in the number of data elements being processed. A standard based on executed operations can represent time efficiency because the number of operations executed is directly related to the time consumed by an algorithm's implementation. Time is not directly considered because it is easier to count operations than it is to calculate expended time.

This standard is not as complicated as it sounds at first. The *ratio change* is the result of dividing an "after" number of operations or data elements by a "before" number. So if the number of processed data elements doubles, the ratio of data-amount change n will equal 2. The relationship between operations p and data amount change n can be expressed mathematically as

$$p = f(n)$$

where $f(n)$ is a number derived from n by some mathematical formula. Common $f(n)$'s include n^2 and n itself. Thus if $f(n)$ equals n^2, then as the number of data elements doubles, the number of operations performed will quadruple. $f(n)$ can also equal a constant, meaning that the number of operations does not change as the number of data elements changes. It is usually too difficult to get an exact value for a constant, so it is just expressed as n^0, which equals the constant 1 or the indeterminate constant c. Constant multipliers to n^2 or other $f(n)$'s are omitted for the same reasons.

The look-up-table algorithm in the sequential construct section is of constant order since it uses indexing to convert a binary byte into two ASCII bytes in just two table accesses. The computational algorithm for the same conversion is also of constant order, since two computations suffice for any byte conversion, but it is a larger constant due to the time consumed by the computations.

$f(n)$ is the measure of an algorithm's time efficiency, and it is represented in *order notation* as $O(f(n))$. In practice, the enclosed $f(n)$ is replaced with a mathematical formula. An algorithm whose number of performed operations increases as the square of the ratio increase in data elements (i.e., whose $f(n)$ equals n^2) is said to be an $O(n^2)$ algorithm. There is a large time penalty for processing numerous data elements with such an algorithm.

To be useful to practicing programmers, the order of an algorithm must be evident by simple inspection of the algorithm. This inspection consists of examining the algorithm's processing structure, be it sequential, selection, or repetition, and observing how the number of mathematical and assignment operations is related to the number of data elements being handled.

Example:

A particular algorithm for sorting names alphabetically runs through a list of x names once for each name in the list while sorting. What is the order of this algorithm?

Since there are x names in the list, the algorithm will look through the list x times. Processing x names x times requires $x \times x$ or x^2 sets of element operations. This algorithm is therefore $O(n^2)$. Thus, doubling the number of names in the list will increase the number of name-handling operations performed by a factor of 4.

We will see an example of algorithm selection based on order shortly.

Several common algorithmic orders are listed in Table 3.13, from the smallest order and best for most purposes on top to the largest order at bottom. Algorithms of adjacent orders can differ in execution speed by hundreds of times with data structures of appreciable size, so pay careful attention to selecting the algorithm of the smallest order on the chart. The possible exception to this rule is that an algorithm of larger order may consume less space, be simpler, or even be faster if only a very small amount of data will ever be processed. You will have to evaluate this on an individual basis. Don't worry about the log2(n) function in the table; it will be defined later in this chapter.

A similar comparison of algorithms for space rather than time efficiency can be made; the order is then one of space rather than time. Whether quick code or compact code is the goal, however, choosing the algorithm of lowest order is by far the most important step in reaching it.

Once the most efficient algorithm has been chosen, additional efficiency can be obtained by implementing it skillfully. This is the purpose of the second stage of optimization.

TABLE 3.13 ALGORITHMIC ORDERS

Rank	Order
Best	constant
↑	log2(n)
	n
	$n \times$ log2(n)
	$n^{3/2}$
	n^2
	n^3
Worst	2^n

Implementation Optimization

To implement an algorithm efficiently one must understand the role of the input and output data in the most basic operations of their processing. Since this understanding is seldom complete from the start, it usually takes several passes to optimize an implementation.

As an algorithm is expanded to obtain the detail required for programming, the role of the data becomes clearer. At the lowest level of pseudocode this role can be seen most clearly, and without the syntax rules of assembly language to distract from it. This is why it is important not to omit the level of pseudocode that translates more or less directly into assembly language instructions.

An understanding of the data can be exploited in at least four ways. First, data that can be most efficiently processed with register-specific CPU operations can be placed into the appropriate registers. Counter and index values are two examples of data that are more efficiently handled in registers. Proper use of the accumulator can also accelerate arithmetic and logical operations.

Second, data can often be placed in page-zero memory for faster access. The appropriate register and memory placement choices are more obvious if you make a list of all data elements and accompany each element with the types of CPU operations to which it will be subjected.

Third, by observing how the available CPU instructions lend themselves to data manipulation in a given situation, the input data can sometimes be redefined in their grouping or content to decrease the number of instructions required to process them. For instance, refer to the checksum example of the repetition construct section to see that the 'numbyt' variable, whose value indicated the number of data bytes in the memory area, could have contained the number of data structure bytes, that number plus one to also account for the numbyt byte, or that number plus two to account for numbyt and for the trailing checksum byte. It turned out that the task was performed most easily and quickly if numbyt held the number of data structure bytes plus two. Although that choice was not pointed out at the time, it was nevertheless consciously made.

The fourth way to exploit an understanding of data is to use the special capabilities of the CPU instruction set knowledgeably with the data. The more exotic addressing modes, like indirect indexing or creative uses of the stack, are often overlooked by programmers but can greatly streamline an algorithmic implementation. Multiplication and division by numbers that can be broken into sums of powers of 2 is often faster using shift and rotate instructions; so a multiplication by 10 can be performed as the sum of a shift-left multiplication by 8 and a shift-left multiplication by 2. Finally, although some will disagree with this statement, a technique called self-modifying code can be used under very special and very restricted circumstances for large time savings.

Self-modifying code refers to the modification of instruction operands by instructions during program execution. Thus a constant loaded as an immediate

operand becomes a variable if its location in memory can be modified during program execution.

The objection to self-modifying code is that it makes a program's functioning difficult to understand or predict. The argument is that if operands vary dynamically as the program is executed, it is difficult to understand how the program really works. This argument is valid in almost all circumstances. The one exception is when the modification is performed by and on instructions within a repetition construct at a low program level, and the repetition executes a set number of times until its purpose is accomplished as opposed to executing continuously until some outside event occurs. In this case the context of the modification is so restricted that if the modifying instructions and the modified operand are adequately commented, there will be no mistaking their purpose or effect. Indeed, the modification in effect adds an addressing mode to the instruction set.

As an idea of the potential gains from using self-modifying code in the proper context, look over the following example.

Example:

Design a repetition construct that moves multiple 256-byte pages from one memory address to another. Implement it with and without self-modifying code.

One algorithm for this process is

```
;move page_count pages of data from source to destination
      LOOP
            move one page of data from source to destination
            decrease page_count by one
         EXITIF page_count equals zero
            move source address up one page
            move destination address up one page
      ENDLOOP
;END move
```

Aside from pre-LOOP assignments, the only expansion needed to prepare this pseudocode for programming is that of the first line. It is

```
;move one page of data from source to destination
      set byte_counter equal to 0
      LOOP
            move byte from address (source + byte_count)
            to address (destination + byte_count)

            decrement byte_count
         EXITIF byte_count = 0
      ENDLOOP
;END move
```

The following completed template uses no self-modifying code. It assumes that the source and destination addresses are being passed in the locations 'source' and 'destina-

tion', with the least-significant byte first, and that the number of data pages to move is being passed in location 'totpgs'. Because the source and destination addresses are stored as variables, indirect indexed addressing can be used to access them.

```
;move page_count pages of data from source to destination
      ;set page_count equal to total_pages to be moved
        LDX TOTPGS ;keep page_count in X
      ;LOOP
       PGLOOP:
            ;move one page of data from source to destination
                 ;set byte_count equal to 0
                   LDY #$00 ;keep byte_count in Y
                 ;LOOP
                  BTLOOP:
                     ;move byte from source address + byte_count
                     ;to destination address + byte_count
                       LDA (SOURCE),Y
                       STA (DEST),Y
                     ;decrement byte_count
                       DEY
                    ;EXITIF byte_count = 0
                       BNE BTLOOP
                 ;ENDLOOP
            ;decrease page_count by one
              DEX
          ;EXITIF page_count equals zero
            BEQ DONE
            ;move source address up one page
              INC SOURCE + 1
            ;move destination address up one page
              INC DEST + 1
      ;ENDLOOP
        JMP PGLOOP
        DONE:    ;the next construct begins here
;END move
```

In the self-modifying version, the locations 'source' and 'destination' are in the construct itself. They are the operands of the LDA and STA operations in the inner LOOP. The LDA and STA operations can use absolute indexing for speed because the source and destination addresses are passed directly into the data movement instructions. Thus the construct executes as if the construct were always written to move data between only those two addresses. Examine the construct closely to see how this works:

```
;move page_count pages of data from source to destination
    ;set page_count equal to total_pages to be moved
      LDX TOTPGS     ;keep page_count in X
    ;LOOP
     PGLOOP:
```

```
          ;move one page of data from source to destination
              ;set byte_count equal to 0
                LDY #$00    ;keep byte_count in Y
              ;LOOP
               BTLOOP:
                  ;move byte from source address + byte_count
                  ;to destination address + byte_count
                    LOAD:                ;'load' label used with assembler
                    SOURCE = LOAD + 1    ;directive to define address
                                         ;of operand for self-mod.
                      LDA BEGADR,Y       ;'begadr' is a dummy label
                    STORE:               ;used like 'load' to define
                    DEST = STORE + 1     ;self-mod address
                      STA ENDADR,Y       ;'endadr' is a dummy label
                  ;decrement byte_count
                    DEY
                ;EXITIF byte_count = 0
                  BNE BTLOOP
              ;ENDLOOP
          ;decrease page_count by one
            DEX
        ;EXITIF page_count equals zero
          BEQ done
          ;move source address up one page
            INC SOURCE + 1
          ;move destination address up one page
            INC DEST + 1
    ;ENDLOOP
       JMP PGLOOP
       DONE:    ;the next construct begins here
;END move
```

As you can see, only two instructions were changed from one version to the next. However, those two instructions were at the heart of the tight inner LOOP moving data from the source area to the destination area. This loop executes every time a byte is moved.

In that loop there are an LDA, an STA, a DEY, and a BNE. The only difference between the two construct versions is the addressing mode of the LDA and the STA. Looking up the "*t*-state" or "clock time" consumption for these operations and addressing modes reveals that the non-self-modifying inner LOOP takes 16 *t* states for most executions, while the self-modifying inner LOOP takes 14 *t* states. This is a 12.5% speed improvement on the inner LOOP, where the bulk of construct execution time occurs. Further, the bulk of a program's execution time tends to occur in this sort of tight loop, so the improvement is likely to affect greatly the overall program speed. Although this improvement is nothing like that available from choosing an appropriate algorithm, it is still a worthwhile improvement to achieve for so little effort at the end of the implementation stage.

Optimization Summary

The optimization techniques that we have discussed can be grouped and summarized by the construct categories they affect. These groupings are:

Sequences: Algorithm selection; efficient instruction set, register, and memory map usage.

Selections: All sequence techniques; look-up-table replacement of CASE constructs having parallel data movement options.

Repetitions: All sequence techniques; self-modifying code in LOOPs of restricted scope.

A more substantial example using both stages of optimization to develop a finished construct can now be given.

Example:

Write a speed-efficient construct that multiplies two binary numbers together.

We will develop an algorithm based on a study of the arithmetic of multiplication. To provide symbolic labels for each number in a multiplication problem, the names for each data element in a multiplication problem are labeled as follows:

```
    multiplicand
×   multiplier
------------
    product
```

The simplest way to solve a multiplication problem is to add the multiplicand to itself "multiplier" times. Because of the wide range of value a multiplier can take on, however, an algorithm based on this method will execute a number of operations proportional to the size of the multiplier. The order of such an algorithm is $O(n)$, where n represents the multiplier size as the number of integer values between 0 and the value of the multiplier.

The widely varying execution time of such an algorithm with different multiplier values causes us to look for a better algorithm. Since a pencil-and-paper "long multiplication" can solve the problem "123 × 123" in about the same length of time as the problem "789 × 789," it seems that there should be a method of computer multiplication that is similarly insensitive to the specific values being multiplied. Indeed, such a method can be directly adapted from long multiplication. Observe the following long multiplication:

```
    456
  × 301
  -----
    456      partial product
+ 000        partial product
+ 1368       partial product
-------
  137256     product
```

The long-multiplication method uses one single-digit multiplication plus one addition per multiplier digit, instead of the conceptually simpler multiplier additions of the previous method. The long-multiplication method can be made more repetitive by adding the partial products into a running sum as they are generated.

First observe the decimal-number version of the long-multiplication algorithm:

```
;multiply two decimal numbers together
    set 'running_sum' to 0
    starting with rightmost digit of the multiplier:
    LOOP
        set 'partial_product' equal to the product of the
        'multiplier' digit and the multiplicand

        add 'partial_product' to 'running_sum'

        shift 'multiplicand' left one place (i.e.,
        multiply it by 10)

      EXITIF leftmost 'multiplier' digit has been used to
          obtain the next-left 'multiplier' digit
    ENDLOOP
    done ('running_sum' holds product)
;END multiply
```

Note that the multiplicand is multiplied by 10 on each pass through the loop to compensate for the increasing place values of the multiplier digits, so that they can always be treated as individual digits. In paper-and-pencil long multiplication we do something similar; each multiplier digit is treated as a single-digit multiplier, but the resulting partial product is shifted-left one place before it is added to the others.

The long-multiplication algorithm becomes even simpler when adapted for binary numbers. Since the only allowable binary digits are 0 and 1, the 'partial_product' of each internal LOOP multiplication will equal either 0 or the multiplicand itself. This means that the binary form of this algorithm can be performed without using any multiplication operations whatsoever. The available CPU operations are then sufficient for performing all aspects of this algorithm.

Because of the machine details of the CPU, it turns out that it is slightly faster to shift the running sum one position right instead of shifting the multiplicand one position left. Shifting the running sum right is equivalent to the hand operation mentioned above, of shifting the partial product one place left, as long as the bit positions of the final running sum are interpreted with the proper weights or place values.

The final form of the long-multiplication algorithm for binary numbers is

```
;multiply two binary numbers together
    set 'running_sum' to 0
    starting with rightmost digit of multiplier:
    LOOP
        IF 'multiplier' bit equals 1
            THEN add 'multiplicand' to 'running_sum'
```

```
            ENDIF
            shift 'running_sum' one place right
        EXITIF leftmost 'multiplier' bit has been used
            obtain the next-left 'multiplier' bit
    ENDLOOP
    done ('running_sum' holds the result)
;END multiply
```

This algorithm, which we'll call the *shift-add algorithm,* performs at most one addition for every bit in the multiplier. The logarithm of a value is equal to the number of times that its number base, 2 in this case, must be multiplied by itself to equal the value. In other words, the base-2 logarithm of n, $\log2(n)$, is the power of 2 that produces n. With a little reflection you can see that the logarithm of an m-digit number in any base is between $m - 1$ and m. Thus the shift-add algorithm, which performs at most m additions on an m-digit multiplier equaling n, is an $O(\log2(n))$ algorithm.

For a one-byte multiplier, the shift-add algorithm requires up to 8 additions, while the repeated-addition algorithm requires up to 256 additions. You can see the advantage an $O(\log2(n))$ algorithm such as shift-add multiplication has over an $O(n)$ algorithm such as repeated-add multiplication.

The shift-add algorithm can now be expanded. Most of the lines involve shifts, adds, and bit tests. There are assembly language instructions for each of these operations, so the pseudocode is nearly detailed enough to program from. The second stage of optimization must now be considered.

Recall that the role of data in the algorithm must be identified to optimize the implementation. The input data elements are the multiplier and multiplicand, and the output data element is the running sum.

The next step in optimization is to identify the roles of the multiplier, multiplicand, and running sum data elements in the processing. Their placement in registers and memory can then be chosen for the greatest efficiency. According to the pseudocode, these data elements have the following roles:

1. The multiplier is bit-tested.
2. The multiplicand is added.
3. The running sum is shifted and added.

If we restrict the multiplier and multiplicand to 8 bits in length, both will fit in single registers. The running sum for an 8×8 multiply can be up to 9 bits long, so it requires two bytes of storage.

These facts allow us to choose storage locations. The most restricted role is taken by the running sum, which must be placed where it can both be shifted and added. There is only one location that both of those operations can access and leave results in; that is the accumulator. Therefore, we will keep the lower part of the running sum in the accumulator. The upper part of the running sum will be kept in page-zero memory so that bits can be quickly shifted from the lower to the upper byte (recall that shift operations can address both the accumulator and memory).

The multiplicand is added to the running sum, which means that it must be placed where the addition operation can access both. Since only memory locations fit that requirement, the multiplicand will be placed in page-zero memory for fastest access.

Finally, the multiplier must be shifted to obtain its bits from right to left. This indicates using the LSR operation repeatedly to place the bits from right to left into the carry flag for testing and use. Only memory locations and the accumulator can be addressed by the LSR operation, so with the accumulator taken, the multiplier must also go into page-zero memory.

The EXITIF line indicates that the multiplication has been completed when the leftmost bit in the multiplier has been used. The easiest way to know this has happened is to count bit shifts and tests until eight (for an 8 × 8 multiply) have occurred. This is easily done by loading the X register with the number 8 and decrementing it each time through the loop until it reaches 0.

The location assignments we have made are:

Number	Location
'multiplier'	Page-zero memory
'multiplicand'	Page-zero memory
'running sum'	Accumulator + page-zero memory
'bit counter'	X register

Plugging these assignments into the pseudocode yields

```
;multiply two 8-bit binary numbers together
    ;set 'running_sum' to 0
        load 0 into A and running sum memory location
    ;starting with the rightmost digit of the multiplier:
        load X with the bit counter value 8
        shift 'multiplier' right one place into carry flag
    LOOP
          ;IF 'multiplier' bit equals 1
            branch past ENDIF if carry = 0
                THEN add 'multiplicand' to 'running sum'
          ENDIF
          ;shift 'running_sum' one place right
            rotate accumulator right into carry
            rotate memory location right to pick up carry
      ;EXITIF last 'multiplier' bit has been used
        decrement 'X' and if result is zero branch to 'done'
          ;obtain the next-left 'multiplier' bit
            shift 'multiplier' right one place into carry flag
    ENDLOOP
    done
END multiply
```

One implementation optimization is to consolidate the initial and final shift multiplier right lines as the first line in the LOOP: this takes care of both the LOOP setup, and the following LOOP executions. This requires starting right of the rightmost multiplier bit. Further, it makes the EXITIF the last statement in the LOOP, so we can

use its conditional BRANCH operation for the usual JMP at the ENDLOOP as well. Finally, we will add an instruction at the end to place the accumulator byte of 'running_sum' into page-zero memory. With these revisions the completed construct can now be written. Pseudocode lines that translate directly into assembly language without adding any additional information are omitted.

```
;ASSIGNMENTS
     BITCNT = 8          ;number of bits in multiplier
     * = $80             ;memory data start at 80h
     RUNSUM * = * + 2    ;'running_sum'
     MULTCD * = * + 1    ;'multiplicand'
     MULTPL * = * + 1    ;'multiplier'
;END ASSIGNMENTS

;multiply two 8-bit binary numbers together
     ;set 'running_sum' to 0
          LDA #0
          STA RUNSUM
     ;starting right of the rightmost multiplier bit:
          LDX #BITCNT
     ;LOOP:
      MLLOOP:
                ;obtain the next-left multiplier bit
                 LSR MULTPL
                ;IF 'multiplier' bit equals 1
                     BCC SHFSUM
                     ;THEN add 'multiplicand' to 'running_sum'
                          CLC
                          ADC MULTCD
                ;ENDIF
                ;shift 'running_sum' one place right
                 SHFSUM: ROR A
                           ROR RUNSUM
           ;EXITIF last 'multiplier' bit has been used
             DEX
             BNE MLLOOP     ;go on to next bit if any left
       ;ENDLOOP
          STA runsum + 1      ;save top byte of running sum
;END multiply
```

Division is the inverse operation of multiplication. We will complete this example by providing you with the corresponding division algorithm. You can implement it using the principles we have discussed in this chapter.

The standard long-division problem accepts a dividend and a divisor as input data, and produces a quotient and a remainder. The relationship between these quantities is

$$\frac{\text{dividend}}{\text{divisor}} = \text{quotient} + \frac{\text{remainder}}{\text{divisor}}$$

We will write the algorithm for a 16-bit dividend, an 8-bit divisor, and a 16-bit quotient. As the quantity being divided, the dividend roughly corresponds to the running sum quantity produced by the multiplication. The final running sum was stored in two memory locations and used the accumulator for intermediate storage and calculation, so the dividend will reverse this pattern and start out in two memory locations and also use the accumulator for intermediate storage and calculation. However, the division algorithm leaves the remainder in the accumulator when it is finished.

Like the multiplier and multiplicand of the multiplication operation, the divisor and quotient will be stored in memory locations.

Reversing the logic of the multiplication algorithm produces a shift-subtract algorithm for multiplication. It encloses the central LOOP with an IF test for division by 0, which of course was not needed for multiplication. The algorithm is given below for your study or use.

```
;divide two binary numbers
  set quotient to 0
  IF divisor <>0
    THEN
      LOOP
          shift dividend left 1 place into accum
          IF divisor <= accum
              THEN subtract divisor from accum
                   put 1 in quotient's LSBit
          ENDIF
        EXITIF last dividend bit has been placed in A
          shift quotient left one place
      ENDLOOP
    ELSE
      set 'cannot divide by zero' indicator (byte, msg, etc.)
  ENDIF
;END divide
```

After the execution of the algorithm either the remainder is in A and the quotient is in memory, or the "cannot divide by zero" indicator has been set.

Exercise:

Use the following questions to review optimization.

(y) Which stage of optimization provides the largest efficiency gains?

(z) How do you estimate the order of an algorithm?

(aa) Name three ways of optimizing an implementation.

Once optimized code has been written, it must be tested to verify that it works properly under a variety of expected and unexpected circumstances. This testing process is our next topic.

TESTING

All three structural levels of a program must be tested, but testing begins at the construct level. To test a construct fully, the construct would have to be run with all possible legal and illegal combinations of input and output data values. With even a small construct such as the one in the multiplication example of the preceding section, it could take months to run all the test cases that would have to be executed for total testing. Running all the test cases for an entire module can literally take centuries.

A more sensible approach is to identify a complete set of categories for all types of input and output values, and to run a representative test of each category. Six major categories cover this need sufficiently, although only a subset of them may apply to any given construct.

The six categories to test for are:

1. Normal output data
2. Boundary input data
3. Boundary output data
4. Invalid input data
5. Invalid output data
6. Input data that exercise a particular execution path

The procedures for running sufficient tests in each category are described below.

1. *Normal output data.* Select a typical output value (or values if multiple data elements are produced by the construct) for each legal value range or type, and try to obtain it from all significant combinations of typical input values that should produce it. A typical value is a value that is in a legal range of values and that is not otherwise distinguished from its surrounding values. So a construct that performs a simple transformation on either positive or negative values should be tested under this category twice: once for normal positive output and once for normal negative output.

In the multiplication example of the preceding section, a reasonable test case would be a number that is the product of two prime numbers. The two possible combinations of typical input values would be with one prime as multiplier and one as multiplicand, and then to reverse their roles as multiplier and multiplicand.

2. *Boundary input data.* Select all input values that are both legal and yet distinguished from their surrounding values, and run the construct to test for the expected output values.

The value 0 is often considered a boundary value, when it is not illegal as in the divisor of the division algorithm, for instance. 0 is a boundary input value for the multiplier and multiplicand of the multiplication example of the preceding section.

The largest and smallest values allowed by a construct are also boundary

values. Other boundary values can be determined by examining the specific construct.

3. *Boundary output data.* Select all output boundary values and, as in category 1, test all significant combinations of input values expected to produce each output value.

4. *Invalid input data.* Select representative input values that are illegal to a construct's processing, and run the construct to see if the expected results are obtained (the result should be some type of error indication and recovery from the error). Illegal data are usually values that are meaningless or out of bounds to the construct's processing. Any construct in which such an error occurs should be changed in structure to allow an error trap execution path that reacts appropriately to the invalid data. The division algorithm contained an error trap for division by zero; it used a selection construct to allow it to react to either valid or invalid input data.

5. *Invalid output data.* This category requires a thought test rather than a code execution. Imagine possible categories of invalid outputs and examine the construct to see if data in any of them can be produced. An invalid output is an output that is neither a normal valid output nor an error message. The processing that allows any such values should then be corrected at the highest level of pseudocode that the processing mistake was introduced.

6. *Input data that exercise an execution path.* The primary purpose of some constructs is to select between actions rather than to transform input data into output data. The first five test categories may not exercise all execution paths in such a construct. For instance, if the construct has a CASE structure that selects different actions for each of several normal input values, none of the foregoing categories will reliably exercise all the CASE options. Such a construct should be fully exercised by selecting a set of input data that will activate every execution path in the construct. The programmers should determine and look for the expected output of each test case.

With every construct you test, make your best effort to identify inputs and outputs in each category. Satisfy yourself that a category truly does not apply to a construct before you move on.

The particulars of each test case you run must be written down to ensure that the test is thorough and that its relationship to the other test results is clear. The information you need to record includes the test category, its input and output data, the expected results, and the actual results when the test is run. To keep yourself honest, be sure and write down the expected results *before* running the test. The scorecard sheet shown in Fig. 3.21 is a useful format for recording this information and for examining it afterward.

A successful test is a test that finds an error. A test that finds no errors does not guarantee error-free code; it simply has failed to unearth errors that may still exist. So, as you design test cases, try to use your intuition within the testing categories listed above to pick the highest-risk cases and increase the likelihood of successful testing.

Test Score Card

Test Subject:

Test Category	Input Case	Expected Output	Actual Output

Test Categories

I) Normal output data
II) Boundary input data
III) Boundary output data
IV) Invalid input data
V) Invalid output data
VI) Input data that exercises a particular execution path

Figure 3.21

Testing for errors and correcting them is called *debugging*. A program to assist in object code testing, called a *debugger,* was probably included in your assembler purchase. Machine-language monitors do not include a debugger, but they include many debugger functions. Debuggers allow you to place data of your choice in memory or in registers, to execute a program from any address you choose, to ex-

ecute them one step at a time and then observe the register and memory contents between each instruction execution, and to execute instructions in the range between a start and stop address, to name a few of the most useful functions.

As a construct or module is executed with test cases, two types of errors will become apparent. There will be errors due to incorrect assembly language notation. This type of error includes among other things the use of the wrong number-base or addressing-mode emblems, mistyped labels, and the use of the incorrect mnemonic for a given operation. Most of these errors are caught by the assembler before object code is produced. This type of error can be corrected by changing individual instructions in the completed assembly language template.

A more serious type of error includes those where the algorithm itself or some expansion of it works differently than desired. These errors must be corrected at the pseudocode level in which they were introduced. The pseudocode must then be reexpanded as in the initial construct development to produce a new version of the construct.

Some of the most common but easiest-to-fix errors of this type occur in the condition-testing instructions of selections and loops (i.e., in IFs, CASE comparisons, and EXITIFs). If the instructions test only for the expected values of a variable, and the construct has execution options only for the expected values of the variable, an unexpected value of tested variable can cause undesired and erroneous results.

To prevent this from happening, a technique called *error trapping* is used. Every type of value a condition variable can take on is tested for and reacted to with a safe, meaningful construct action. So even those values that seemingly should never occur during construct execution will be met with an error message, a prompt for a new user input, or some other graceful response. Of course, if an erroneous condition-variable value ever does occur, the construct or constructs causing that value should be corrected.

Example:

Illustrate the effects of using and not using error trapping.

Consider a program that calculates the amount of dynamite needed to blast a tunnel section without caving in the rest of the tunnel. Assume that a particular CASE construct in the program selects from its calculating options by testing a simple selection variable named 'rock_type'. Rock_type has three possible values: 'igneous', 'sedimentary', and 'metamorphic', which have been defined as the values 0, 1, and 2, respectively.

The CASE construct could test 'rock_type' for the values 0 and 1, and assume that if 'rock_type' equaled neither of these, it must equal 2. This seems like a reasonable assumption, and saves writing a few assembly language instructions.

However, assume 'rock_type' is assigned its value by another construct as the program runs. If the assigning construct is faulty, it might place a value other than 0, 1, or 2 in the variable. For instance, it might set bit d7 of the value temporarily for some reason and fail to reset the bit. Thus the intended value 1 would be passed as the value 81, the CASE construct would test for values 0 and 1 and then interpret the "81" as if it

were 2 or 'metamorphic', and the user would get seemingly valid but, in fact, faulty results. If the program reached the field, a geology professor or experienced ordinance person might catch the error, but someone less skilled might follow the program's recommendations and destroy an entire tunnel in one blast!

Simple error trapping would prevent this from happening. The CASE construct would test 'rock_type' for the values 0, 1, and 2, and assume that if 'rock_type' equaled none of these, there was an error. The fourth execution option, for an erroneous rock_type value, would print an error message and allow the programmer to find the error before the program reached the field.

Module-level testing is similar to construct testing, except that testing category 6) is no longer applicable. Testing the hierarchical organization is an intellectual exercise à la test 5, and a topic for the next chapter.

Exercise:

Answer the following questions to review the testing process.

(bb) What are the differences between normal, boundary, and invalid test cases?

(cc) What is the difference between a syntactic error and a logical error? What is the difference in how they are corrected?

Chapter 4 introduces the heart of programming, which is the dividing of a proposed task into module-sized functions, and the designing of a structure that will coordinate the identified functions to perform the original task. After learning those techniques the programmer will be ready to use the programming techniques of this chapter to produce significant programs. However, he or she will still need the I/O techniques of Chapters 5 to 7 to make use of the special abilities of the Commodore 64.

FOR FURTHER STUDY

Subject	Book	Publisher
Constructs	*Top-Down Structured Programming Techniques* by C. L. McGowan and J. R. Kelly	Petrocelli-Charter
Algorithms	*The Art of Computer Programming,* Vols. I, II, and III; by D. Knuth	Addison-Wesley
Testing	*Software Testing Techniques* by Boris Beizer	Van Nostrand Reinhold

4

IMPOSING REASON: PROGRAM PLANNING

When the first electronic computer was built in 1946, its memory and processing resources were so scant and unsymmetric that programming it required every trick its users could devise. From then until 1966, programming was viewed as the means to perform successively larger tasks using clever shortcuts pulled from hard-won bags of tricks. The programmer spent a large portion of his or her time trying to make small pieces of the program run microseconds faster or occupy a handful fewer bytes. This style of programming is still practiced by many programmers, whom we now call "hackers."

By 1966, the drawbacks of programming in this unsystematic and small-scale manner were attracting the attention of a few widely scattered people. One result was the proof that three basic processing structures, the sequence, the selection, and the repetition, were sufficient for all programming needs.

In 1968, the need for an overall structured way of programming was generally recognized for the first time when the NATO allies called a conference on "software engineering." The unreliability of computer programs had begun to threaten the security of nations.

This heightened attention had a most important result: People finally realized that the task performed by the programmer in developing software is distinct from the task performed by the software itself, and that software development must be studied as a separate discipline to learn its secrets. This distinction has been understood for centuries by artisans in other creative fields. For instance, authors, composers, and visual artists have long applied a few compositional laws to produce works of many styles, meanings, and historical periods.

The laws for developing high-quality software, like the laws for developing

quality literary, musical, and artistic works before, had to be observed or deduced, tested, and then recorded. Such laws, and the techniques they spawn, release one's creativity by expanding the scope of the work that one can create and by minimizing one's unprofitable efforts.

Since programming was at that time still caught up in small-scale thinking, the first laws sought for were those that concerned the coding of individual instructions in a program. The principle that a program should have some sort of structure evolved into two laws: the law that a program should be divided into modules, and the law that program structures should be limited to the three basic construct patterns. Upon these laws the phase of software development known as *structured programming* was built. Many at that time embraced structured programming as the final solution to the problems of software development. As with the earlier "bag of tricks" style of software development, structured programming has become an end in itself for many programmers. The laws and techniques of structured programming were discussed in Chapter 3, although they were never labeled as such. We will resummarize them shortly.

Unfortunately, programmers using just these techniques and laws found that although their constructs and modules were easy to write and understand, those structures often failed to cooperate with each other in performing an overall task. The resulting programs frequently failed to run at all. The need to discover higher laws of software development became obvious.

By 1975 the major laws governing the design of the highest structural level of a program, the hierarchy, had been identified. These laws became the basis for the phase of software development known as *structured design.*

Programs written according to the principles of structured design and structured programming were easy to develop and to understand, usually worked when completed, but often performed the task incorrectly! There was still no reliable way to divide a task into the right subtasks, the subtasks that when performed together would completely perform the original task. Of course, this meant that there was no good way to select modules for an intended program. Even so, most programmers today practice no more than structured design and structured programming.

By 1977, laws sufficient for dividing tasks reliably had finally been identified. These laws were the basis for the phase of software development called *structured analysis.*

The term "structured" introduces the name of each phase because each phase is conducted along a well-defined pathway of techniques and laws. The programmer is guided through the entire software development cycle by a structure of activities that if followed faithfully will lead to a program that accomplishes everything the programmer originally intended. This assumes, of course, that the programmer's intentions are within the physical limitations of the computer.

In this type of software development the programmer starts with structured analysis, follows with structured design, and finishes with structured programming. Structured programming can be viewed as the implementing of a program model derived from structured analysis and structured design. The latter two phases are

parallel to the two phases of data modeling discussed in Chapter 2. Let's look at the three phases from this perspective.

As Chapter 2 explained, the general name for the first phase of modeling is *analysis*. Analysis identifies the purest form of a system among its sometimes confusing surroundings, divides it into its fundamental parts, and defines what each part must do in the overall system. This defines the requirements that any copy of a system must meet to make it equivalent to the original system, so analysis is sometimes called *requirements definition*. Analysis can be applied to both real and imaginary systems, but in programming the system to be analyzed is usually an imaginary one that the programmer wishes to implement.

The structured analysis phase identifies the extent of a task, identifies its component subtasks and their data interfaces, and defines what the subtasks do. In a similar manner, data selection identifies the scope of information needed to perform a task, identifies its component data elements, and defines their role in the task.

Chapter 2 also noted that the general name for the second phase of modeling is *synthesis*. Synthesis recombines the parts defined by analysis into an efficient model of the original system. This model is thorough enough to implement an actual system from. Synthesis is often called *design*.

The structured design phase synthesizes component subtasks to produce an efficient model for performing an overall task. This model embodies the top level of program structure, the hierarchy. Similarly, data structuring synthesizes data elements to produce a model for an efficiently processed data structure. As indicated in Chapter 2, data structures can, but need not, be hierarchical.

Finally, as we have already said, structured programming implements the design model into the working system called a *program*.

We will be studying these phases in reverse order, in the order that they were developed instead of the order that they are normally used, so that you will be able to use the phases as you learn them. This order also puts the role of each phase in clearer perspective.

Therefore, the body of this chapter begins with a summary of structured programming, with which you are already familiar. Next is the discussion of structured design. Finally, structured analysis is covered. Together, these three development phases will provide a complete method for utilizing your computer to perform any task within its memory, speed, and I/O capabilities. The Commodore 64's memory and speed capabilities were discussed in Chapters 1 and 3. Its I/O capabilities will be discussed in Chapters 5, 6, and 7.

STRUCTURED PROGRAMMING

The principles of structured programming were thoroughly explored in Chapter 3, so we will just summarize them here.

The six techniques of structured programming are:

1. Writing levels of pseudocode, starting with an optimal algorithm for a module
2. Using flowcharts, tables, or other aids in developing the pseudocode
3. Optimizing the pseudocode implementation
4. Completing templates with assembly language instructions
5. Testing the templates
6. Correcting the level of pseudocode that each error was introduced at, all levels below it, and finally the templates themselves

The two laws of structured programming are:

1. *Modularity.* Programs should be divided into *modules* that perform the different functions of the overall program task.
2. *Structured code.* Program instructions should be organized into one of three processing structures: the sequence, the selection, or the repetition.

These principles and techniques were illustrated with many examples in Chapter 3, so we will omit further examples here.

We are now familiar with all the activities of structured programming. Before these activities can begin, a model of the hierarchical module structure must be made from which to program. This model defines all the modules needed in the program and their interrelationships. From this model each module can be written using structured programming. When all the modules shown in the model have been written, and they have been tested both individually and as a group, the program will be complete.

Developing a model of a hierarchy of modules that performs a given task in the most efficient manner possible is the goal of structured design.

STRUCTURED DESIGN

According to *Webster's,* to *design* is to "plot out the shape and disposition of the parts"* that will go into a product. The plotting is typically carried out on a model of the product. By this definition, the shape and disposition of model parts are the most important aspects of a design. This definition strongly implies that the *shaping* and *placing* of those parts are the two most important design activities.

A familiar illustration of design is found in the automotive industry. An automobile designer knows from the start that the car he or she will design must include certain parts, such as an engine, a transmission, brakes, and a body. These parts and their relationships have been identified by prior analyses. The auto

**Webster's Third New International Dictionary,* G. & C. Merriam Company, Springfield, Mass., 1971.

designer designs the car by shaping the imaginary parts and placing them in a model of the physical car. This model has traditionally been kept as drawings on paper, but is increasingly being stored as computer files that can be viewed and altered from video terminals.

In software design, the component parts are the basic stand-alone subtasks defined by analysis. Also defined by analysis are the data flows needed between subtasks for the overall task to be completed. At the beginning of the design process the subtasks are notated in a hierarchy chart as modules that perform those subtasks. The duties of these modules will be adjusted and refined during the design process as the model is perfected.

Parameter-passing connections between modules, called *interfaces,* are the offspring of the data flows identified between subtasks during the structured analysis phase. We shall see examples of all these things shortly.

The programmer designs a program by shaping and placing the prospective modules into a model of the most efficient way to perform the overall task. Modules are *shaped* by shifting their inner functions between them. Modules are *placed* by setting them into a hierarchical framework and by moving them around within that hierarchy to alter their working relationships. The details of shaping and placing will be discussed shortly.

Program models are still usually kept on paper, although they will be stored and manipulated on computers as software is written to support those functions.

As we have said, making an optimal program model is the goal of structured design. Therefore, our first concern in studying structured design is to understand what the program model is.

The Program Model

A program model consists of three items. One of these items is the central component of the model, and the other two elaborate on the first. The first item is the *module hierarchy chart,* and the latter two are the *data dictionary* and the *module specifications.* After we describe each of these items we will illustrate it with an example.

Module hierarchy chart. The module hierarchy chart is the heart of the program model. It shows modules in an organizing framework with information passing between them. These chart elements are discussed separately below.

Modules. Recall from Chapter 3 that a hierarchy chart represents modules as boxes. Each box contains a short description of the module's actions, in the now familiar "imperative verb . . . object" format.

Organizing Framework. The module boxes are placed in rows or "levels" in the chart. Any two module boxes on adjacent levels can be connected by a line to signify that the upper module utilizes or *invokes* the lower one to perform part of its overall task. During a structured programming phase using assembly language, all

such connections will be implemented with a calling JSR instruction in the upper module, and a returning RET instruction at the end of the lower module. However, the means by which the upper module uses the lower is irrelevant to the design process, so hierarchy charts are equally useful in programming with languages that do not use JSRs and RETs.

Module Communications. Two types of information are passed between modules and are therefore shown on hierarchy charts; they are data interfaces, which represent the coded information whose transformation is the main purpose of a program, and *control interfaces,* which represent coded information sent by one module to affect what, when, and how another module works.

The assembly language implementation of an interface between two modules has one module write values into predetermined memory locations or registers, and the other module remove the values from the same locations. Again, such programming details are not considered during the design process.

A data interface consists of a data structure which can be as simple as a single variable or as complex as a multileveled data structure (e.g., a sequence built of repetitions built of selections, and so on). A control interface almost always consists of a simple variable with several possible values in a selection structure.

Data interfaces are the data passed between modules for transformation or output. They are an inherent part of performing an information-handling task, and the data flows they are based on are therefore identified during the analysis phase. We have discussed data and data handling extensively in preceding chapters, so we will move on to the basics of module control and control interfaces.

The most important aspect of module control is provided by the hierarchical framework; each module above the lowest level of the chart can invoke the modules connected one level beneath it to perform its major subtasks. These modules manage the modules beneath them by controlling when they execute. Modules in the lowest level of the chart have nothing to manage. Instead, they perform the basic data transformations and movements of the overall task. The lowermost modules work rather than manage.

The top module in a hierarchy is usually a pure manager, while the lowermost modules are pure workers. In between modules are frequently part manager and part worker, with the relative proportion of these duties depending on the depth of the module in the hierarchy levels.

Manager modules do not normally require control interfaces to utilize their workers. In a well-designed module hierarchy, the only control a manager exercises over a worker is to start it executing. When the worker is done, it returns its data results and execution control to the manager.

However, an imperfect design can make it necessary for a managing module to issue direct commands to its subordinates to control what, when, or how they do their jobs. Or, even in a good design a managing module may need information about the outcome of its subordinates' tasks for it to decide on its own next action. These two situations require two different types of control interfaces.

Control interfaces of the first type are called *command interfaces,* for obvious reasons. By issuing such commands, the manager becomes intimately involved with the details of each worker's job. This has several disadvantages. The manager's overall function becomes diluted with bits and pieces of functions lifted from its subordinate modules. The dismembered jobs of the workers are never given the isolated programmer attention necessary to make sure that they have been totally understood and accounted for. The programmer has to consider the same job details in more than one module, and must make sure that all modules over which a job is spread will work together as expected.

Command variables have a selection organization, with each value option representing one way to do the job. Such a variable might, for instance, command one of several possible destinations for the worker module to send its data results to, or command one of several possible actions for the worker to carry out.

Command interfaces do not belong in polished hierarchy charts. Eliminating them is a part of the design process.

Control interfaces of the second type are called *status interfaces,* because worker modules use them to tell their managers what happened when they did their jobs (i.e., to give their managers a status report). For instance, if a manager module invokes a subordinate whose job is to search a disk directory for a user-requested file, when the subordinate is finished the manager will want to know whether or not the file was found. This information cannot be considered a data interface because it is not transformed into any part or stage of the data that the program will produce. It is a control interface that the manager uses to decide what to do next. Since it does not command any action of the manager, it is not carried by a command variable. It is instead carried by *a status variable.*

Status variables, like command variables, have a selection structure. Their value options represent the different possible types of outcomes of a module's task. If there is only one type of job outcome, or if different types of outcomes are adequately described by the module's data interfaces, status variables are unnecessary.

The manager module receiving a status variable need not know how the sending worker module produced it. Thus most of the disadvantages that come with command interfaces do not apply to status interfaces. Nevertheless, status interfaces are minimized in a good design model. This is because a good design places the most closely related functions together in their own modules, and the information needed for decision making can then be obtained within the module making the decision.

All command interfaces and most status interfaces are *implementation dependent;* that is, they will vary between different programs written to perform the same task. Therefore, they are usually identified during the design phase as the model's need for them is recognized. Again, a control interface passing downward is carried by a command variable and marks a flawed design, while a control interface passing upward is carried by a status variable and indicates a possible area of design improvement.

An interface is shown in a hierarchy chart with its name beside an arrow pointing from its source to its destination. The arrow's tail indicates the type of informa-

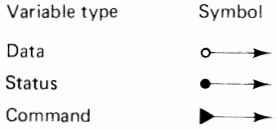


Figure 4.1

tion passed; an outline-circle tail represents a data structure, a filled-circle tail represents a status variable, and a filled-triangle tail represents a command variable. These symbols are shown in Fig. 4.1.

The following example shows a module hierarchy chart for a task of moderate complexity. Study it carefully to ensure that you understand the broad functioning of the system it is describing. This task will be the subject of examples throughout this chapter. Although workable, the chart shown below is intentionally imperfect and overcomplicated so that it can be dramatically improved later using the techniques and laws of structured design.

Example:

Show a module hierarchy chart for a computer system that manages a personal telephone directory.

The hierarchy chart in Fig. 4.2 shows the modules and data interfaces whose subtasks and data flows were previously identified in an analysis phase which we will not show. It also shows the control interfaces necessary for the chart to model a working system.

Remember that the name of the top module in a hierarchy is also the name of the overall hierarchy. Thus this hierarchy is named 'manage_directory'.

According to the hierarchy chart, 'update_directory' invokes 'add_listing' and 'delete_listing' at various times in performing its own job, and so on with the other connected modules.

The chart implies a decision to let 'accept_inputs' wait as long as necessary to get a valid user input. This eliminates the need for a status variable telling 'manage_directory' that there have been no valid inputs.

The 'job_type' variable is needed to select one of the three jobs this system can perform: updating listing, retrieving numbers, and transferring the directory. This interface is similar to a command variable, since it selects an action, but it differs in one important way. The receiving module, 'manage_directory', requests the control information and chooses the options it will accept. Thus 'manage_directory' is the active solicitor of self-defined decision-making information, not the passive recipient of an externally defined command. It is like the difference between an owner of a company taking a customer's order for an item in the company line and an employee receiving an assignment from the owner. This difference makes user command information more like a status interface than a command interface.

The other user-input status variables are needed to tell 'update_directory' whether to add or delete a listing, and 'transfer_directory' the direction of transfer. Note that these variables are passed through 'manage_directory' to the two second-level modules. Later on we will see that this is undesirable, and indeed, that it shows an opportunity to improve on the design. The remaining interfaces are the data used by the different job types. 'Update_directory' must know the name in a listing to delete the

listing, and must know both a name and a phone number to add a listing. 'Retrieve_number' must know the name in a listing. The 'name' and 'number' data interfaces provide this information.

The module hierarchy chart provides a complete overview of a system that performs a particular task. Detailed definitions of the interfaces and modules are kept in the supporting data dictionary and module specifications.

Data dictionary. The word "data" in *data dictionary* should be taken in the larger sense of all coded information, since both data and control interfaces are defined in the data dictionary. These definitions specify both the internal structure and the information contents of each interface. Any data structures accessed solely by worker modules and therefore not passed on module interfaces are considered system interfaces and must also be defined in the data dictionary.

The data definitions are notated in the data dictionary format explained in Chapter 2. To review, the sequence structure is shown with the plus sign, the selection structure with brackets enclosing data separated by vertical lines, the repetition structure with enclosing braces preceded by the number of the fewest possible repetitions and followed by the number of the most possible repetitions, and comments with enclosing parentheses and asterisks.

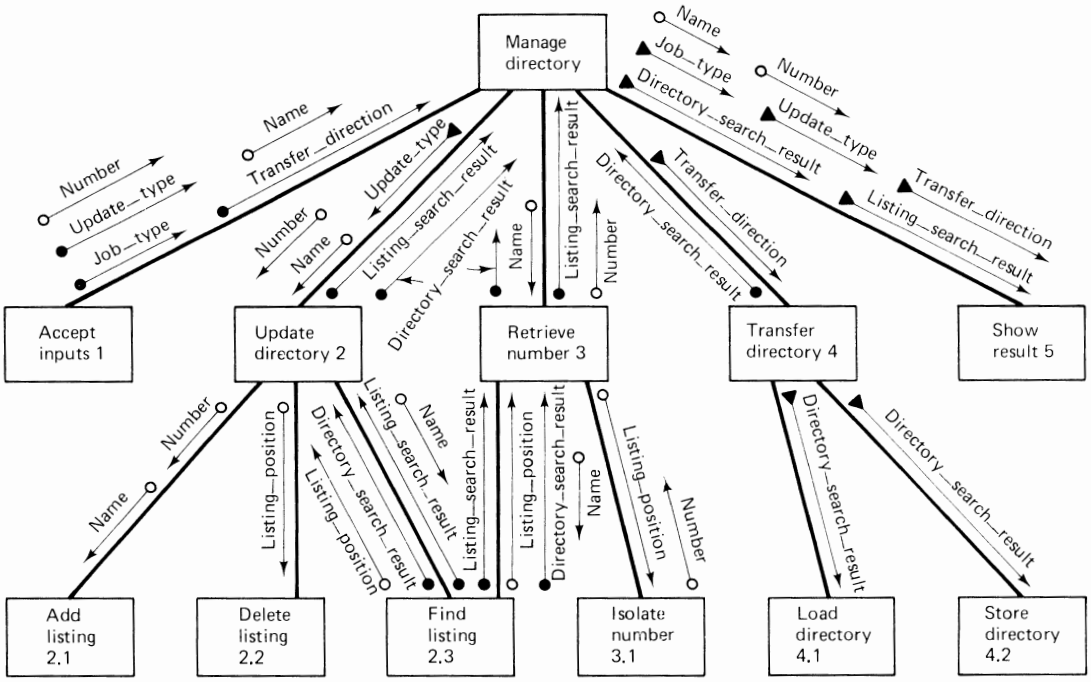


Figure 4.2

Example:

Give the definitions for the data and control interfaces in the 'manage_directory' hierarchy.

Assume that the directory is defined as being from 1 to 100 listings in length, where each listing contains a person's name and telephone number. Each 'name' data structure is 10 bytes long, to hold up to ten ASCII characters. The name put in the data structure will be left-justified (i.e., the first letter of the name will be in the first byte of the data structure, and the data structure will be padded with ASCII space characters as needed after the name to fill out the 10 bytes). Assume also that phone numbers are always 10 digits long, so that every phone number includes an area code. Preceding the first listing is a byte whose value equals the total number of listings in the directory at the time. This directory will only be accessed by the lowest-level worker modules, which is why it is not named as an interface in the hierarchy chart. It is defined in the data dictionary as a system interface.

The definitions are listed alphabetically to make them easier to locate. They are

```
directory                = directory_size + 1{listing}100
directory_search_result  = [ found : not_found ]
directory_size           = (*number of listings in directory
                             as binary value from 1 and 100*)
listing                  = name + number
listing_position         = POINTER TO listing (*listing position
                             as binary value from 1 to 100*)
listing_search_result    = [ found : not_found ]
job_type                 = [ update : retrieve : transfer :
                             quit ]
name                     = 10{(*ASCII letter*)}10 (*left-
                             justified and padded w/ trailing
                             ASCII space characters*)
number                   = 10{(*ASCII digit*)}10
transfer_direction       = [ load : store ]
update_type              = [ add : delete ]
```

Pay close attention to the 'directory' definition, since it plays a particularly important role in the example in the next section.

Module specifications. The module specifications define exactly what the modules must do as their part of the overall task. These definitions are written as pseudocode processing descriptions. From them the algorithms of structured programming are developed as explained in Chapter 3. The transition from module specification to top-level programming algorithm can be as simple as using the specification verbatim. It is a question of efficiency. The specification describes what is to be done; the top-level algorithm says how it will be done for greatest efficiency. Modules that are too simple or too limited to reward by choosing a more complicated algorithm for performing their tasks should be programmed in the most direct manner from their specifications.

Example:

Show the module specifications for the modules in the 'manage_directory' hierarchy.

These specifications are given below. You must refer to the hierarchy chart and data dictionary to understand fully the internal workings of these modules. Make sure that you grasp these specifications, for they will be seen again later in the design process.

You should note that the user's input options have been intentionally and sometimes significantly limited for the sake of simplicity. For instance, once a 'job_type' has been selected there are no opportunities for the user to "bail out" without giving all the inputs for that job and seeing it to its completion. Such options are very desirable in real programs, but also add considerably to their complexity. Note also that specific algorithms for significant module functions, such as searching the directory, are not a part of the module specifications. Algorithm selection belongs in the programming phase and not in design. Finally, remember that this entire model, including these module specifications, is unimproved so that we can practice the techniques and laws of structured design on it later. Most of the complexity in this model is due to its unimproved condition, and the eventual improved model will be much simpler.

```
'manage_directory';manage the use of the phone directory
  LOOP
      accept_inputs
    EXITIF job_type = 'quit'
      CASE job_type OF
        transfer: transfer_directory
        retrieve: retrieve_number
        update:   update_directory
      ENDCASE
      show_result
  ENDLOOP
END 'manage_directory'
```

'Manage_directory' has overall responsibility for accepting user inputs, exiting the program, performing any of the three major program functions, and showing the results to the user. To carry out these responsibilities, 'manage_directory' invokes (e.g., CALLs) subordinate modules to perform the major subtasks. The specifications as they now exist for those modules and their own subordinates are shown below.

```
(1)   'accept_inputs' ;requests and accepts user inputs
        get job_type from user
        CASE job_type OF
          transfer:  get transfer_direction from user
          update:    get update_type from user
                     get listing name from user
                     IF update_type = add
                       THEN get listing number from user
                     ENDIF
          retrieve:  get listing name from user
          quit:      (empty option: take no other action here)
        ENDCASE
      END 'accept_inputs'
```

```
(2)   'update_directory' ;add or delete a directory listing
        find_listing
        CASE update_type OF
          add:      IF directory_search_result = not_found
                      THEN create an empty directory in memory
                    ENDIF
                    IF listing_search_result = not_found
                      THEN add_listing
                      ELSE delete_listing
                           add_listing
                    ENDIF
          delete:   IF listing_search_result = found
                      THEN delete_listing
                    ENDIF
        ENDCASE
      END 'update_directory'

(2.1) 'add_listing' ;add a listing to the end of the directory
        IF directory_size < 100
          THEN make a listing from name and number
               add the listing to the directory
               add 1 to directory_size
        ENDIF
      END 'add_listing'

(2.2) 'delete_listing' ;remove a listing and recompact the directory
        delete listing pointed to by listing_position
        move all following listings up one listing position
        subtract 1 from directory_size
      END 'delete_listing'

(2.3) 'find_listing' ;find listing containing name
        set directory_search_result = not_found
        set listing_search_result = not_found
        search for directory on disk
        IF directory is found
          THEN set directory_search_result = found
               search directory for listing containing name
               IF listing is found
                 THEN set listing_search_result = found
                      set listing_position = listing position (1 to 100)
             ENDIF
        ENDIF
      END 'find_listing'

(3)   'retrieve_number' ;retrieves phone number if listing is present
        find_listing
```

```
        IF listing_search_result = found
          THEN isolate_number
        ENDIF
      END 'retrieve_number'

(3.1) 'isolate_number' ;isolate phone number from listing
        set number = phone number in listing at listing_position
      END 'isolate_number'

(4)   'transfer_directory' ;move the directory between disk and memory
        set directory_search_result = not_found
        search for directory on disk
        IF directory is found
          THEN set directory_search_result = found
        ENDIF
        CASE transfer_direction OF
          load:  IF directory_search_result = found
                   THEN load_directory
                 ENDIF
          store: IF directory is not in memory
                   THEN tell user there is no directory to store
                   ELSE store_directory
                 ENDIF
        ENDCASE
      END 'transfer_directory'

(4.1)   'load_directory' ;load the directory from disk to memory
          load directory from disk into memory
        END 'load_directory'

(4.2)   'store_directory' ;store the directory from memory to disk
          IF directory_search_result = found
            THEN delete the existing directory
          ENDIF
          store directory from memory to disk
        END 'store_directory'

(5)     'show_result' ;display results of executing job_type
          IF directory_search_result = found
            THEN CASE job_type OF
                   update:   IF listing_search_result = not_found AND
                               update_type = delete
                                 THEN show 'no listing to delete'
                                 ELSE show 'directory updated'
                             ENDIF
                   retrieve: IF listing_search_result = found
                               THEN show number on TV
                               ELSE show 'no number listed'
                             ENDIF
```

```
            transfer:  IF transfer_direction = load
                         THEN show 'directory loaded'
                         ELSE IF directory not in memory
                                  THEN show 'no directory to store'
                                  ELSE show 'directory stored'
                              ENDIF
                       ENDIF
            quit:      show 'quitting program'
        ENDCASE
        ELSE CASE job_type OF
            update:    IF update_type = add
                         THEN show 'directory updated'
                         ELSE show 'directory not found'
                       ENDIF
            retrieve:  show 'directory not found'
            transfer:  IF transfer_direction = store
                         THEN show 'directory stored'
                         ELSE show 'directory not found'
                       ENDIF
            quit:      show 'quitting program'
        ENDCASE
    ENDIF
END 'show_result'
```

Programming these modules would require selecting algorithms and developing their pseudocode to the template level of detail. Programming the I/O operations (i.e., the data movement to and from disk, TV, and keyboard) requires knowledge of the computer's operating system. This information is supplied in Chapter 5.

The algorithms for most of these modules will be straightforward. However, a few modules, such as 'find_listing', which searches the directory for individual listings, require a more significant algorithm choice during structured programming. For instance, the simplest search algorithm for 'find_listing' is a front-to-back comparison of each directory listing against the desired directory name. The time taken by this algorithm depends on how listings are added to the directory (another algorithm choice). Assuming that the listings are in no particular order, the average time taken by a sequential search of n listings would be proportional to $n/2$. This is because the average length of all searches is halfway through the directory. This algorithm with no data ordering has order n performance, since multiplied or divided constants such as the 2 in $n/2$ are ignored in algorithmic order. The order of this algorithm can be decreased (i.e., its speed improved) by placing the listings in the directory in decreasing order of usage. Thus the most-often-accessed listing will be first, the second-most will be second, and so on. The simplest way to achieve this is to prompt the user to input listings in this order, but listings added later will be out of order or will require additional program logic to place them in their proper position. In this application unordered data elements are sufficient. Thus listings can simply be added to the end of the directory until the directory is full. However, this algorithm with frequency-ordered data, or more advanced search algorithms of order down to $n \times \log2(n)$, would be desirable with a larger

directory or a more demanding search problem. The most generally useful of the advanced search algorithms are discussed in the follow-up volume *Beyond Power Programming*.

With the components of a program model understood, we can now discuss program design.

The Design Process

Program design is a two-stage process. First, the programmer places the parts defined by structured analysis into an initial program model. Then he or she perfects the model using the techniques and laws of structured design. We will discuss these two stages separately.

Creating the initial model. The programmer creates the first version of a program model directly from the results of the structured analysis phase. As we shall see in the structured analysis section, the results of analysis are pictures and text defining the basic parts of a task. Recall that these parts are the separate subtasks and their data interfaces.

The analysis results are transformed into a design model with a technique called *transform analysis*. This name misleadingly implies that the technique is part of analysis; nevertheless, the name is widely accepted and we will use it also. The model produced by transform analysis is complete except for control interfaces, which are filled in by the programmer after careful thought. A supplementary technique called *state analysis* assists in this process and is discussed in *Beyond Power Programming*. However, it is not necessary for effective programming.

Since transform analysis works directly with the results of structured analysis, we must postpone studying it until we have discussed the structured analysis phase.

Once an initial model has been obtained, we can begin the work of perfecting it into an efficient and trouble-free basis for structured programming. This is the purpose of the second stage of structured design.

Perfecting the model. The second and concluding stage of the design process perfects an initial hierarchy into an optimal model of how to perform an overall task. This is done by using the remaining techniques of structured design to change the model, and the laws of structured design to guide the changes and check the quality of the results. We will discuss the techniques first and then the laws.

The Techniques of Structured Design. Recall that design is the plotting out of the shape and disposition of parts that have been identified by analysis. After the parts are placed into an initial model using transform analysis, the model is perfected with techniques for reshaping the parts and replacing them within the hierarchy. We will call the first of these techniques *re-forming the modules*, and the second *reorganizing the hierarchy*. These two techniques are discussed below.

Re-forming the Modules. The first technique is to divide the functions in a module specification and redistribute some or all of them among other modules in the hierarchy. This specifically allows new modules to be created as long as the functions within them existed before in one or more other modules. However, placing the new modules in the hierarchy is part of the second technique.

Reorganizing the Hierarchy. The second technique is to move, add, or remove modules in the hierarchy. Like the prince and the pauper, the manager can become the worker and the worker the manager. Alternatively, a module may simply be moved to another branch of the hierarchy tree at the same level. As the modules are moved they will also usually need to be re-formed, since their responsibilities to other modules will have changed.

These two techniques are the only ways to improve a design model. They are used when the laws of structured design reveal defects in the program model. We will discuss these laws now.

The Laws of Structured Design. These laws take the form of rules that a program model should obey. Violations of a given law reveal specific weaknesses in the program design. Each type of weakness has a specific solution through the techniques of structured design or by backing up to the step before the error was introduced. Therefore, the discussion of each law includes an explanation of the law, the weaknesses it reveals, and the techniques used to cure those weaknesses. The section on each law is followed by an example applying that law to the 'manage_directory' model to reveal any specific weaknesses. After all the laws have been discussed, a final example will show the perfected 'manage_directory' model.

The laws are discussed in their general order of importance, although the ordering can only be approximate. There are nine laws, the first of which is called *completeness.*

Completeness. The first law of structured design states that "the program model must show the desired task being performed correctly and completely." A model may violate this law by performing only part of its original task, or by performing the original task incorrectly and thus producing undesired results. Careful inspection of the module specifications, starting at the top level of the hierarchy chart and proceeding down, is the most likely way to reveal violations of this law.

There are only two reasons for this error to appear in a model. Either the analysis used to produce the model was incorrect, or the techniques of structured design have been used incorrectly and have altered the basic nature of the model.

How this problem is corrected depends on how it arose. If it is the result of a faulty analysis, the only solution is to start over with another analysis phase. This is not so bad; programming is a repetitive activity, with the programmer gradually closing in on the desired result. It cannot and really should not be a one-pass activity, since many of the best features of programs are serendipitous discoveries from improving them.

If the violation of this law is caused by the incorrect use of design techniques, the solution is much easier. If the programmer keeps copies of each generation of the design, he can simply back up to the last version of the model before the mistake was made and continue from there.

Example:

Apply the law of completeness to the 'manage_directory' model.

The 'manage_directory' model as supplied performs the desired directory management task completely.

Conservation of Data. The second law of structured design states that "all module input data must be used to produce module output data, and all module output data must have corresponding module input data from which they are derived." In other words, a module cannot create data from nothing, and data cannot simply disappear into a module without a reason. Of course, the lowest-level modules of a hierarchy chart can have system interfaces to outside devices, and so on, making the modules appear to violate this law. However, studying the module specifications will reveal the outside source of these data items.

Violations of this law show up in the hierarchy chart and sometimes in the module specifications. It is easy on the chart to show data arrows pouring out of modules that have no way of producing them, and it is equally easy to show data arrows disappearing into modules without a trace. However, the written specifications for those modules will either not match their pictures in the hierarchy chart—that is, they will not produce the impossible data outputs and they will show no mention of the disappearing data inputs—or the module specifications will be muddled on how those data items are handled.

In the first case, where the hierarchy chart and specifications do not match, there is either a problem in the original analysis, with the transform analysis, or with the incorrect use of the design techniques. These problems are corrected as with completeness violations; the programmer backs up as far as necessary to get a correct basis to work from. In the second case, where the problem is with muddled module specifications, the only cure is to return to the analysis phase. Module specifications are directly obtained from subtask specifications written during analysis.

Example:

Apply the law of data conservation to the 'manage_directory' hierarchy.

The supplied initial model obeys the law of conservation of data. 'Accept_inputs' and 'show_result', which on the surface appear to violate this law, have system interfaces that show up in their module specifications.

Coupling. The third law of structured design states that "module interfaces should be as few and as nonintrusive as possible." Interfaces are sometimes called *couples,* hence the name of the law.

The more this law is violated, the less the modules look like black boxes, and the more complicated the module specifications and general model functioning become.

Violations of this law are revealed on the hierarchy chart and confirmed in the module specifications. As you already know, there are three types of interfaces: data, command, and status. Every command interface in a hierarchy is a violation of this law. Every status interface should be viewed as a possible violation. Even multiple data interfaces passing in one direction between two modules should be viewed suspiciously.

Modules connected by few data interfaces and no control interfaces are said to be *loosely coupled.* Modules connected by many interfaces are *tightly coupled.* The benefits of loose coupling include obtaining black-box modules and simplifying program testing and debugging. The drawbacks of tight coupling include the scattering of jobs over multiple modules with the extra code and complexity needed to make the pieces work together. Also, the complicated interactions of the tightly coupled modules require much more time to test and may still conceal untested bugs afterward.

Too much coupling or coupling of the wrong kind indicates that related functions have been scattered between modules, and should be regrouped so that they are together in the same modules. This is done by re-forming the modules.

There are variations on the three interface types that yield a number of different possible coupling categories. An interface that seems to belong to two or more of these categories is classed with the worst of them. This is because connected modules are no more independent than their tightest coupling. Because the different kinds of coupling vary in their desirability, we will discuss them individually from the best or loosest coupling to the worst or tightest coupling. You should identify all the interfaces in your designs by their coupling type to identify problem modules and the corrective actions needed.

The best type of couple is an indivisible data item. This type of interface is simply called a *data couple.*

Example:

Identify the data couples in the 'manage_directory' model.

From the data dictionary definitions we know that these interfaces are 'name' and 'number' (logically, names and phone numbers seem more like single low-level entities than like collections of characters).

If a data structure is passed instead of an indivisible data item, the interface is called a *stamp couple.* Stamp coupling is nearly as good as data coupling, as long as the data in the structure naturally belong together. Do not group or bundle otherwise independent data into data structures just to draw fewer interface arrows on the hierarchy chart. This only disguises the real problem of too many data couples between modules.

Example:

Identify the stamp couples in the 'manage_directory' model.

The data dictionary shows three stamp couples: 'listing_position', 'directory', and 'listing'. 'Listing_position' is considered to be a stamp couple because it passes a pointer to a data structure instead of to an indivisible data item.

Control interfaces are less desirable than data-item or data-structure interfaces because any kind of control linkage decreases the independence of the linked modules. Nevertheless, status interfaces are preferable to command interfaces because a status interface informs a module without reaching into it and controlling its internal functions. Relatively strong black-box modules can be written with status interfaces.

A status interface is called a *status couple.* Too many status couples between modules in the hierarchy chart shows that related functions have been scattered and should be regrouped so that they are together in the same modules.

Example:

Identify the status couples in the 'manage_directory' model.

The hierarchy chart and data dictionary show 'job_type', 'directory_search _result', 'listing_search_result', 'transfer_direction', and 'update_type' as status couples. Their many appearances on the chart indicate a need to re-form almost every module in the design, but especially in 'accept_inputs'. The reason the latter module requires so many status interfaces, and the key to redistributing its functions to eliminate those interfaces, is found in the fourth design law, which we will study after the law of coupling.

Sometimes an interface will appear repeatedly on a hierarchy chart and its contents be altered by two or more different modules. All modules sending or receiving that interface are said to be *common coupled.* Interestingly, this type of coupling can connect modules that are not adjacent in the hierarchy chart. Such modules are scarcely related except through the common couple and can affect each other through that couple in unexpected and undesired ways. For instance, an error in a module altering that couple can affect *all* modules reading that couple and throw suspicion on all other modules writing to it. It can take a long time to locate and fix such an error.

Common couples are evident on hierarchy charts where two or more modules independently produce interfaces having the same names. Common couples via system interfaces, which do not show up on hierarchy charts, can be found by examining the module specifications. In the final program, the coded information for all interfaces of the same name will be kept in the same physical location, which is usually in memory, but can be kept on mass storage such as disk or cassette by using the operating system modules described in Chapter 5.

There are a few natural and desirable uses for common coupling. In general, any program whose central purpose is to handle one or more major data structures

can justify using common coupling. A chess program is an excellent example of this. The chess board is stored in memory as a data structure that is the focal point of the program. Therefore, the entire program should be able to access it. In this type of program the complication required to avoid common coupling would be worse than the common coupling itself. However, make sure that you have one of these special cases before resorting to common coupling. Then follow all the other laws of design to minimize any resulting problems.

Example:

Identify the common couples in the 'manage_directory' model.

The major common couple is 'directory'. It is not shown on the hierarchy chart because it is a system interface, but the data dictionary defines it and the module specifications show its memory copy being written to by the 'add_listing', 'delete_listing', and 'update_directory' modules, and being read by the 'find_listing' and 'isolate_number' modules. Since handling the directory is the central purpose of the 'manage_directory' model, this common couple is acceptable.

Another common couple is 'directory_search_result'. It is altered by both the 'find_listing' and 'transfer_directory' modules. Since handling this control interface is not the central purpose of the model, this common couple is a design flaw. Note that 'directory_search_result' was also listed as a status couple. Common coupling is worse than status coupling, so by our earlier rule the 'directory_search_result' interface is no better than a common couple. The final common couple is 'number', which carries the phone number input by the user, as well as the phone number found in the directory data structure by module 'isolate_number'.

Coupling with command interfaces is the last type of coupling we will mention and the only totally unacceptable one. We have already explained the problems with such *command coupling,* especially that it fragments subtasks and is incompatible with black-box modules. Command couples are obvious on the hierarchy chart by their triangular tails.

Example:

Identify all command couples in the 'manage_directory' model.

In this initial design, all the status couples are used elsewhere as command couples. This is especially seen at the 'show_result' module. As with the 'accept_inputs' module earlier, the key to re-forming 'show_result' to correct this problem is found in the next design law. For now, though, all the status couples must be downgraded to command couples.

From best to worst, then, the coupling categories are data, stamp, status, common, and command. If your data interfaces are few and of the top two coupling categories only, if you minimize the use of status coupling, if you reserve common coupling for tasks revolving around a central data object, and if you remove all command coupling, your programs will be very loosely coupled with all the resulting benefits.

One other symptom of poor coupling are interfaces that pass through one or

more modules without being processed (i.e., transformed, checked for errors, and so on). These interfaces are called *tramp data,* for they "tramp" through the modules between the initial source and the eventual destination. Passing an interface through nonintervening modules indicates an even greater separation of related functions than is found in adjacent modules with too many interfaces.

Nevertheless, an interface passing from a subordinate module through its manager to another subordinate on its own level is not considered tramp data, since the modules should be black boxes that are unaware of each other, with the manager their only logical link. We say "logical link" because in an actual assembly language program the manager does nothing with the interface; the source and destination modules directly access the same physical location to write or read the information passing between them. This is also true for source and destination modules passing tramp data; the in-between modules do nothing with the interface.

Tramp data are eliminated by reorganizing the hierarchy so that the source and destination modules are adjacent to each other, or by re-forming the source and destination modules to eliminate the need for any interfaces at all.

Coupling is one of the most important design laws. However, it is sometimes difficult to see the best way to correct a model that violates coupling. The remaining design laws are more limited but also more specific in identifying the exact defect in the design model. Of course, once the defect has been identified, the solution is straightforward. In this context the law of coupling is used to identify problem areas in the model, and the remaining laws are used to correct them quickly and precisely.

Example:

Identify tramp-data and any remaining coupling defects in the 'manage_directory' model.

The 'directory_search_result' and 'listing_search_result' interfaces from the 'find_listing' module tramp through 'update_listings', 'retrieve_number', and 'manage_directory' to finally arrive at 'show_result'. Also, 'name' and 'number' tramp from the 'accept_inputs' module through 'manage_directory' and 'update_listings' to 'add_entry'. Finally, 6 of the 12 connected module pairs in the hierarchy chart are bridged by three or more interfaces. Even if those interfaces were all data or stamp couples, the modules involved would need to be examined to see if related functions had been divided among them. Of course, in this model most of the interfaces connecting those module pairs are control couples of one type or the other, and are therefore even worse offenders.

Cohesion. A module with few interfaces tends to have a well-defined task consisting only of directly related functions. The relatedness of the functions in a module is called the module *cohesion.* The law of cohesion simply states that "each module should consist only of directly related functions, and as much as possible, all directly related functions should be grouped together in the same modules."

Although a model that obeys the law of coupling tends to obey the law of cohesion, and vice versa, the two are not equivalent. For instance, tramp data can pass through a strongly cohesive module and make it appear tightly coupled to the

modules adjoining it. Each design must be tested for coupling and cohesion to ensure both module independence and module strength.

Coupling problems around modules in the hierarchy chart suggest possible cohesion problems in those modules. The module specifications can then be studied to identify their specific shortcomings. A weakly cohesive module may need command interfaces to help it decide which of its functions to perform, or it may become such a special-purpose mixture of functions that no one can understand its overall purpose. Modules with poor cohesion cannot be black boxes.

The main technique for improving cohesion is to re-form the modules, dividing and redistributing their internal functions to form more cohesive modules. The goal is to maximize the relatedness of the functions within each module in the hierarchy. As usual, re-forming the modules should be supported with reorganizing the hierarchy as changes to the module responsibilities make it necessary.

Just as only a handful of relationships are possible between members of a family (e.g., uncle, cousin, brother), only a handful of relationships are found between the functions in a module. The cohesion of the entire module is determined by the weakest relationship between any of its functions. We will discuss these types of relatedness or cohesion from best or most strongly cohesive to worst or most weakly cohesive. Keep these categories in mind, or at least remember where to look them up, for it is a good idea to identify the cohesion of every module in a design model.

The strongest type of relatedness is called *functional cohesion.* It exists in any module whose functions work together to perform one well-defined module task. A well-defined module task is evidenced by a strong "imperative verb . . . direct object" module name. Indeed, such a strong name alone heavily implies that a module is functionally cohesive, without any reference to the module specification.

If you will insist on finding strong names for your modules, as we have recommended, you will have few problems with weak cohesion. If after your best efforts a module name remains weak, unclear, or noncommittal (e.g., process data), it probably means that the module's task is ill-defined and that its cohesion will be poor. You can either re-form the modules to create better defined modules, or if too many modules have naming problems you may repeat the analysis phase to obtain more cohesive modules in the initial design model.

Example:

Identify the functionally cohesive modules in the 'manage_directory' model.

From the hierarchy chart and module specifications we see that 'manage_directory', 'find_listing', 'add_listing', 'delete_listing', 'retrieve_number', and 'isolate_number' are functionally cohesive, although several of them violate coupling rules by exporting or simply passing too many interfaces.

The next-strongest type of relatedness is called *sequential cohesion.* It exists in any module whose functions execute in sequential order, with the output data from each function becoming the input data for the following function. This type of module has been compared to a section of an assembly line; although the section

completes no overall task, its functions are in the correct order to complete the section's portion of the overall task.

A major difference between this and the previous type of cohesion is that modules of this type cannot be given a concise, single-job name. At best, sequentially cohesive modules have strong names for each major function in the module, with each name separated by the word "and." For instance, a module named 'accept_and_decode_message' is probably sequentially cohesive, with the 'accept_message' function providing its output to the 'decode_message' function.

As we said, a sequentially cohesive module is like a section of an assembly line. If the functions in the other sections or modules contributing to an overall task can be moved into the first module, so that the entire assembly-line task is performed in one place, the resulting module will become functionally cohesive. The multiple function descriptions in the previous module name will be replaced by a single strong name for the overall module task. Alternatively, the individual functions in a sequentially cohesive module may be large enough to be separated into separate modules performing complete subtasks of the overall assembly-line task. If all the functions in the overall task can be made into separate modules, the whole group of them can be placed under a manager module and the hierarchy reorganized to hold the group. All those modules, including the manager, will then be functionally cohesive. As a further advantage, the separated-out and more fundamental black-box functions may be useful to other manager modules in the hierarchy, saving on duplications of similar functions in the hierarchy.

A module that requires a command interface to function properly cannot be functionally cohesive, since at least one of the functions necessary to do its job is in the module above it, but it can still be strongly sequentially cohesive. By re-forming the module to contain the function it needs to make its own decisions, such a module can easily be upgraded to functional cohesion.

Example:

Identify any sequentially cohesive modules in the 'manage_directory' model.

Inspection of the module names and study of the module specifications show that several otherwise strong modules are receiving command interfaces and are therefore sequentially cohesive. Those modules are 'load_directory' and 'store_directory'.

Modules of either functional or sequential cohesion contain only closely related functions. Such modules tend to be easily programmed, easily understood, and loosely coupled. The remaining types of cohesion are the result of grouping less closely related functions, and produce progressively less successful modules.

A module whose functions perform parts of separate tasks, but whose functions all work with the same input or output interfaces, is said to have *communicational cohesion*. Such functions act like modules within a module that are grouped together only for the convenience of using the same interfaces. Any name for such a module must be a combination of the names of its internal functions. For instance, a

chess-program module named 'record_move_and_test_for_check_or_mate' would be communicationally cohesive. Its two functions work with the same data structure, the chess board. Of course, any other functions that access the chessboard could be grouped to form the same kind of module.

If the functions in a communicationally cohesive module are trivial, the module can fairly safely be left intact. However, if the functions are at all significant, it is better to divide them to produce separate and more strongly cohesive modules, and then to deal with any inappropriate common coupling. Of course, common coupling in a chess program is totally appropriate, making the separation in that case even simpler.

Example:

Identify any 'manage_directory' modules having communicational cohesion.

Inspection of the module names in the hierarchy chart and further study of the module specifications show that there are no communicationally cohesive modules in the design model.

A module whose functions are related only by their occurrence at the same stage of the program's execution is said to have *temporal cohesion.* This type of cohesion is usually found in setup or *initialization* modules, and cleanup or *finalization* modules. Initialization modules do things like setting up initial variable values, getting initial user inputs, and checking to make sure all the necessary I/O devices are attached and working. Finalization modules do things like displaying the program results for the user and turning off I/O devices used by the program.

The functions in a module of this type are usually closely related to tasks performed by other modules. They are grouped in temporal modules only because the designer is thinking temporally and not functionally. Because the functions in the separated tasks have been split among two or more modules, more interfaces are required between the modules and the entire model is harder to understand. There are no advantages to temporal cohesion.

The cure for temporal cohesion is simply to distribute the grouped functions to those modules whose tasks they contribute to.

Example:

Identify the temporally cohesive modules in the 'manage_directory' model.

'Accept_inputs' fits our definition of an 'initialization' module, and 'show_result' qualifies as a 'finalization' module. In both cases their internal functions have been grouped solely to make them all able to execute at the same time in the program loop. Note that the functions in both modules really belong in other modules; for instance, the only module that needs to use the 'number' interface is 'add_listing', so that is where 'number' should be obtained.

'Update_directory' is also temporally cohesive, since its first function of creating a directory in memory is placed there only to make it execute just before 'add_listing'. The function of creating a directory when no directory is available on disk is much more related to the function of loading the directory, so it belongs with 'load_directory'.

We will redistribute all these functions to their proper locations when we have completed discussing the law of cohesion.

A module whose functions seem to be related by subject matter, but which actually contribute to completely separate tasks, is said to have *logical cohesion.* The name is ironic since such cohesion is deeply *illogical.*

Logical cohesion is very dangerous because it appears to have cohesive strength when it actually has none. Because of their apparent strength such modules may be left alone when they should be re-formed, and the result will be extra interfaces and weaker cohesion in all the modules whose related functions have been left in the logically cohesive module.

Consider, for example, the logically cohesive module named 'handle_finances' (whatever that name means). Assume that 'handle_finances' contains the following functions:

1. Balance checkbook.
2. Compute dollar principal in mortgage payment.
3. Estimate risk on currently held stock options.
4. Project effect of national debt on auto interest rates.
5. Generate belligerent letter to bill collectors.

What these functions seem to have in common is "finances". What they actually have in common is . . . nothing!

A logically cohesive module will be coupled so tightly to other modules that changing it slightly can cause the whole program to break down. Such programs are also very difficult to debug.

Example:

Identify the logically cohesive modules in the 'manage_directory' model.

'Transfer_directory' is logically cohesive because its functions of loading and storing the directory are actually related to the task of managing the directory, which is the responsibility of 'manage_directory', and they are grouped only because of their apparent relationship of directory movement. The poor cohesion of this module is evidenced by the command couples in and out of it.

If the functions in a module not only have nothing in common but also do not even appear to have anything in common, the module is said to have *coincidental cohesion.* The only good thing that can be said about coincidental cohesion is that if you analyze a problem at all, and even attempt to give your modules strong names, it will never appear in your programs. As usual, the cure for it is to re-form the offending modules.

Example:

Identify any coincidentally cohesive modules in the 'manage_directory' model.

None of the 'manage_directory' modules have coincidental cohesion.

In summary, the cohesion types from best to worst are functional, sequential, communicational, temporal, logical, and coincidental. The first four laws of structured design reveal most of the defects in a design model. At this point in the design process it is a good idea to use the techniques of structured design to fix all identified flaws. Once these flaws have been eliminated, the resulting design model can be further corrected and fine-tuned from testing the model against the remaining laws of structured design.

Example:

Use the techniques of structured design to improve the 'manage_directory' defects identified by the laws of coupling and cohesion.

We have noted the temporal cohesion of 'accept_inputs' and 'show_results', so we will break up those modules and distribute their functions to the related modules. For instance, the function of displaying the phone number is moved into the 'isolate_number' module that obtains the number. The resulting module can be renamed 'display_listing_number' to reflect its overall function.

Several modules can be moved within the hierarchy to place them closer to their related tasks, and therefore to save on control coupling and tramp data. By their functions we can see that 'load_directory' and 'store_directory' are really needed only at the very beginning and the very end of program execution; they should directly support 'manage_directory'. Therefore, 'transfer_directory' is unnecessary and can be eliminated. 'Load_directory' can be given the function of searching for the directory, previously in 'find_listing', and the function of creating an empty directory in memory, from 'update_directory', to form a module that does whatever it takes to get a copy of the directory into memory. We will rename it 'obtain_directory'.

Similarly, 'add_listing' and 'delete_listing' are direct parts of managing the directory, and therefore 'update_directory' can be eliminated for the same reasons as 'transfer_directory' was before. The verbs "transfer" and "update" in the names of those modules are weak compared to the verbs such as "store" and "delete" in their subordinates' names, which hints at the weakness of the two manager modules. In both cases, moving the subordinate modules to places where they can be directly controlled eliminates the control couples that were needed before.

We will also separate out the functions 'get_listing_name' and 'get_listing_number' so that 'get_listing_name' can be used by the three modules that require that function. The resulting hierarchy appears in Fig. 4.3. Note that 'get_listing_name' is called by two levels of modules. This is perfectly acceptable.

We see that these changes have eliminated all the command couples and all but one status couple. The tramp data are also gone. If any tramp data had been left, we would have applied the techniques of structured design again to pull the source and destination functions into the same or at least adjacent modules. After just one correction stage we now have a smaller and simpler model with fewer interfaces of which all are data or data structures. Further, the model now defines a more logical and understandable method for performing the 'manage_directory' task. Also note that most of the data are passed in the lower levels of the chart, a healthy sign.

The updated data dictionary contains the following definitions:

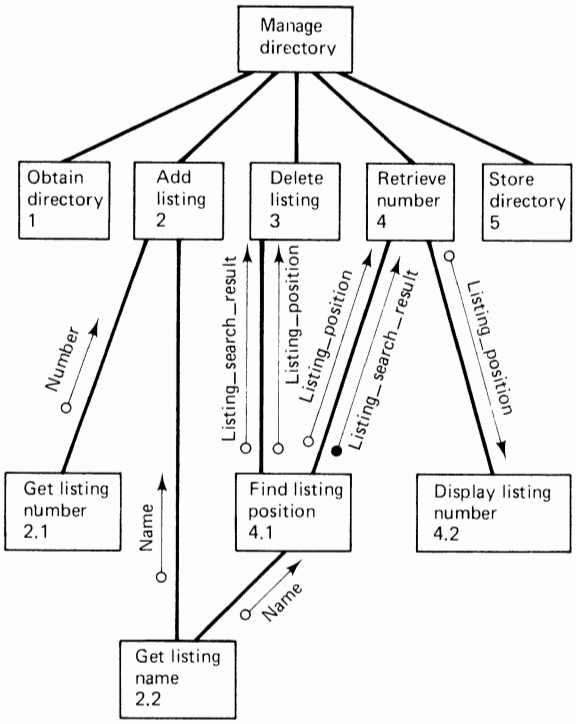


Figure 4.3

```
directory              = directory_size + 1{listing}100
directory_size         = (*number of listings in directory
                           as binary value from 1 to 100*)
job_type               = [ add : delete : retrieve : quit ]
listing                = name + number
listing_position       = POINTER TO listing (*listing position
                           as binary value from 1 to 100*)
listing_search_
    result             = [ found : not_found ]
name                   = 10{(*ASCII letter*)}10 (*left-
                           justified and padded with trailing spaces)
number                 = 10{(*ASCII digit*)}10
```

The module descriptions now read:

```
'manage_directory' ;manage the use of the phone directory
  obtain_directory
  LOOP
      get job_type from user
```

```
        EXIT IF job_type = quit
          CASE job_type OF:
            add:      add_listing
            delete:   delete_listing
            retrieve: delete_listing
          ENDCASE
      ENDLOOP
      store_directory
    END 'manage_directory'

(1) 'obtain_directory'
      check for directory on disk
      IF directory is present
        THEN load directory from disk into memory
        ELSE create an empty directory in memory
      ENDIF
    END 'obtain_directory'

(2) 'add_listing' ;add a listing to the directory
      IF directory_size < 100
        THEN get_listing_name
             get_listing_number
             make listing from name and number
             add listing to the directory
             add 1 to directory_size
             display 'listing added to directory' message
        ELSE display 'directory too large' message
      ENDIF
    END 'add_listing'

(2.1) 'get_listing_number' ;get new phone number from user
      request 10-digit phone number
      set number = the input ASCII number bytes
    END 'get_listing_number'

(2.2) 'get_listing_name' ;get listing name from user
      request listing name
      set name = the input ASCII letter bytes plus trailing spaces
    END 'get_listing_number'

(3) 'delete_listing' ;delete a listing from the directory

      find_listing_position
      IF listing_search_result = found
        THEN delete listing pointed to by listing_position
             move all following listings up one listing position
             subtract 1 from directory_size
             display          'listing deleted' message
```

```
          ENDIF
        END 'delete_listing'

(4)     'retrieve_number' ;retrieve and display phone number if present

          find_listing_position
          IF listing_search_result = found
            THEN display_listing_number

          ENDIF
        END 'retrieve_number'

(4.1)   'find_listing_position' ;find the listing's directory position
          set listing_search_result = not_found
          'get listing name'
          search for listing containing name
          IF listing is found
            THEN set listing_search_result = found
                 set listing_position = listing position (1 to 100)
            ELSE display '(name) listing not found' message
          ENDIF
        END 'find_listing_position'

(4.2)   'display_listing_number' ;display the number at listing_position
          get and display the phone-number in the listing at the
            listing_position
        END 'display_listing_number'

(5)     'store_directory' ;store the directory from memory to disk
          check for directory on disk
          IF directory is present
            THEN delete directory
          ENDIF
          store directory from memory to disk
        END 'store_directory'
```

Each time a design model is altered it must be checked again for completeness of solution and conservation of data. If neither of these laws has been violated by changes to the design, it is time to apply the remaining laws of structured design to the model.

Balance. The law of balance states that "the data inputs to any module must be of similar complexity to its data outputs." This law spotlights modules whose input data interfaces are either much simpler or much more complex in structure than their output data interfaces. A module whose input is 'company mailing list' and whose output is 'employee zip code' undoubtedly violates the law of balance.

This problem is caused by modules that do too much of the work personally (i.e., "shirt-sleeve managers"). By making any offending module into a manager and pulling some of its major functions below it as subordinates, the data can be transformed in stages instead of all at once. This has the side benefits of making the model easier to understand and to program. Thus curing imbalance expands the hierarchy.

Prevention is always better than cure, and the prevention for imbalance is to conduct a more thorough analysis phase. If the overall task has been properly divided into levels of subtasks with no more than nine subtasks below any one subtask the next level up, there will almost never be a problem with gross differences in data complexity between module levels, and almost never a problem with balance.

Example:

Identify the imbalanced modules in the manage_directory model.

The model shows no great difference in complexity between the input and output data of any module, so balance has not been violated.

Module Size. The law of module size states that "modules should be no shorter than one complete function and no longer than two assembly language pages in length."

Until you have written a few programs it is difficult to judge what will be the programmed size of a module from its module specification. However, the spirit of this law is that modules should be kept relatively short but long enough to do a complete job. The ideal length is from 1/3 to 1 assembly language page in length, including comments, so that it can all be viewed at once. A module longer than two pages is too long to keep under your eyes at one time, making it much more difficult to understand, program, or debug.

Example:

Identify any modules in the manage_directory model that violate the law of module size.

Module size is obeyed by all modules in 'manage_directory'.

Generality. The law of generality states that "design models should be built of modules of the most general function consistent with their intended use and the laws of structured design."

Modules obeying the law of generality can often be used in other programs or by multiple higher-level modules in the program at hand. The goal of generality is to make the module perform the general case of the task it is assigned, rather than specific and arbitrary instances of that task. However, be careful not to expand the generality of a module beyond its intended use, since that will require extra work and error risk for no good reason.

Example:

Identify any modules that exhibit generality in the manage__directory model.

One of the products of correcting the model was a module called 'get__listing__name'. This is a general function that was duplicated in several modules before the model was improved. By identifying it as a general function and putting it in a separate module, we were able to make the design a little more focused and to eliminate the unproductive repetition of that function within the model.

Summary of Structured Design

Structured design uses three techniques and seven laws to produce a program design model in the form of a perfected hierarchy chart, data dictionary, and module specifications.

The three techniques of structured design are:

1. *Transform analysis:* transforming the product of structured analysis into an initial design model. Its use will be fully explained at the end of the structured analysis section.
2. *Re-forming the modules:* redistributing internal module functions among modules.
3. *Reorganizing the hierarchy:* moving, adding, or removing modules in the hierarchy chart.

The seven laws of structured design are:

1. *Completeness.* The program model must show the desired task being performed correctly and completely.
2. *Conservation of data.* All module input data must be used to produce module output data, and all module output data must have corresponding module input data from which it is derived.
3. *Coupling.* Module interfaces should be as few as and as nonintrusive as possible. The types of coupling from best to worst are data, stamp, status, common, and command.
4. *Cohesion.* Each module should consist only of directly related functions, and as much as possible, all directly related functions should be grouped together in the same modules. The types of cohesion from best to worst are functional, sequential, communicational, temporal, and coincidental.
5. *Balance.* The data inputs to any module must be of similar complexity to its data outputs.
6. *Module size.* Modules should be no shorter than one complete function and no longer than two assembly language pages in length.

7. *Generality.* "Design models should be built of modules of the most general function consistent with their intended use and the laws of structured design."

From Design to Programming

After a design model has been perfected, the structured programming phase can begin. One way to start is to program all the modules in the system, including their JSRs and RETURNs, assemble the entire program, and run it to see if it works. This approach is called *big-bang building,* because the program will either work or, more likely, blow up completely.

A better approach is to program the modules from the top of the hierarchy down, plugging them into the hierarchy one at a time and testing the resulting partial program each time to see how it works with the new module. This approach is called *top-down building.*

In top-down building the first module to be programmed is the top module of the hierarchy. Since that module calls the modules on the level beneath it, we must provide simple replacements that return to the calling module and allow it to continue executing. Such replacement modules are called *stubs.* Stubs also set up values for any necessary upward interfaces, with those values either written into the stub assembly code before each test run, or by getting them from the keyboard as the stub executes (see Chapter 5 for the means of doing this). The latter is as complicated as a stub should ever get. Complex I/O and data transformation are left out so that the stubs will be reliable without a lot of programming effort.

Once the top module has been tested and debugged, the stubs below it are replaced one at a time with real modules, and the resulting partial program is tested again. When a stub is replaced by a full module, of course, another layer of stubs must be written to substitute for the modules underneath the new module.

New modules should only be inserted after the existing partial program has been thoroughly tested and debugged. Then any errors that occur after a new module is inserted must be due to that module. Either the new module is faulty, or its interactions with the module or modules calling it has exposed their faults. In either case, finding and correcting the error is much easier than with the big-bang technique.

The top-down building process is continued until the lowermost modules in the model have been inserted, tested, and debugged. At that point the entire program will be working and complete.

A variation on the top-down approach that some like better is called *string building.* String building starts with the top module, as in top-down building, but then follows each string below the top module down to the lowest level. It builds down and then across; top-down builds across and then down. One advantage of string building is that the programmer works with closely related modules as a group instead of hopping from one group to another while working down module rows.

However, string building is more difficult to plan out. Either method works well; try both and use whichever you prefer.

Thus the sequence of software development as we now know it is to analyze a task, to convert the analysis into an initial design model, to perfect that model, to write individual modules according to the principles of structured programming, and to build those modules into a tested and working program. We can now study the first two steps in that sequence: analyzing a task and converting the analysis results into a design model.

STRUCTURED ANALYSIS

Webster's defines *analysis* as the "separation or breaking up of a whole into its fundamental elements or component parts."* This definition applies to software analysis if the whole is a task that we want a computer to perform. Analysis lets us understand a task by identifying the many individual processes that must be performed to complete that task. Those processes are the parts of the whole.

In the design phase we used a set of task-building techniques on a concrete representation of a program. We called the program representation a *program model.* In the analysis phase we will apply a set of task-dividing techniques to a concrete representation of the task we are trying to understand. We will call this task representation a *task image.* Like a program model, a task image consists of pictures and text that can be physically manipulated into an optimal form. However, a task image only describes the parts of a task; it says nothing about how to perform that task with a computer. That is an advantage during the analysis phase since mistakes can be made by designing a system to perform a task before the task is fully understood.

The task image and the structured analysis process allow us to come to a full understanding of a task before we start designing a way to perform it with a computer. How well an analysis of a task is performed sets the limit on how well the program to perform that task will work.

Developing a complete and accurate task image is the goal of structured analysis. Thus we begin our study of structured analysis by discussing the task image.

The Task Image

A task image consists of three items. One of these items is central to the task image and corresponds to the role of the hierarchy chart in the design model. This item is a set of *data flow diagrams;* you saw them before in Chapter 1. The two other items

**Webster's Third New International Dictionary,* G. & C. Merriam Company, Springfield, Mass., 1971.

support the first. One of these is a *data dictionary,* which is identical to the data dictionary of the design process except that it leaves out the implementation details of the low-level data elements (e.g., that characters are ASCII encoded). The other supporting item is a set of *process specifications,* which are in the same form as the module specifications of structured design.

Data flow diagrams. In Chapter 1 we said that data transformers and the data flows between them are the basic elements of all information-handling tasks except those involving switching (see the section "Types of Information Tasks" in Chapter 1). Data flow diagrams, or DFDs, reduce a task to its purest state by representing only its data-handling aspects. Control information (i.e., information used to coordinate or command the execution of data transformers) is specifically excluded from data flow diagrams. Control information is omitted because it concerns *how* the task will be performed, a design issue, rather than *what* the task is, an analysis issue. The only exception to this rule is that status information essential to performing the task can be shown. For instance, status messages to the TV and thus to the user may be necessary. Considering unnecessary control information during analysis locks the analyst into specific ways of dividing a task, instead of allowing him or her the total freedom to discover the most logical way to divide the task.

DFDs show the data-handling processes required to perform a task as if they were occurring simultaneously. Imagine these processes running from the initiation of a task until its completion; one is "on" now, then another, and so on like twinkling lights until the task is completed. This is the way your computer performs a task; one process at a time. If you hold open the shutter of an imaginary "data-sensitive" camera during this entire period, you will have a photograph of the data usage of all processes at all execution times. This is what a DFD does. A DFD allows us to examine every stage of performing a task at once, which is important if we are to understand the task as a whole instead of as isolated pieces. Incidentally, thinking of a task as a group of simultaneous processes is not all that artificial. Computers are now being built that assign a separate CPU to each process so that all the processes can be executed simultaneously for a manyfold speed improvement. This approach, called *parallel processing,* will be used in personal computers before long.

Four distinct types of objects can be shown in DFDs. In Chapter 1 we saw two of these objects: arrows, which represent data flows, and circles, which represent data transforming processes. Additionally, whenever a data flow branches and goes to more than one destination or comes from multiple sources, the branch is shown on a darkened point. You will see this in the next example.

There are two other data-related things that a DFD must show about a task. First, it must show where the task obtains its initial data and where it sends its completed data. These outside data sources and destinations are the context of the task we are interested in. They are called *terminators* and are represented with boxes. Second, it must show data that are created and used at different times, or are used repeatedly while a task is performed. Most data are used just once, right after their creation, but data that are used repeatedly must be stored between usages. They are

kept in data *stores,* which you can think of as data flows at rest. Stores are symbolized by two parallel lines.

All the DFD objects—data flows, processes, terminators, and stores—must be named to identify their role in the task. Data flows, terminators, and stores are "things" and are named with nouns. Processes are actions, so their names are based on verbs in the familiar "imperative verb . . . direct object" format. The data flows into and out of a store need not be named, since they symbolize moving copies of the data structure named in the store.

Example:

Show a DFD from which the final manage_directory design model could have been directly developed.

We developed the final manage_directory model from an inefficient and flawed initial model. We could have instead developed a similar model from a DFD such as the one shown in Fig. 4.4. Note that this DFD shows all the major data-handling operations seen in the 'manage_hierarchy' design model: adding and deleting listings, retrieving phone numbers, and loading and storing the directory. The data used and produced by each of these operations are shown on the data flows that connect with them.

Note also that the DFD omits control information such as the 'listing_search_result' status flag in the design model. The main purpose for such information is to help modules decide when and how they will do their jobs; it does nothing to help us understand what the underlying processes do.

In the later stages of program development, both the data flows and the resting data flows or stores will be implemented as variables and data structures. Stores must be handled more carefully than data flows, however, because they may affect several modules via common coupling, or single modules at different times via a kind of time coupling.

The DFD processes will eventually become design-model modules, and the terminators will show up in module pseudocode when I/O devices such as the keyboard and the video screen or TV set are accessed.

There are a few rules that govern how the four types of DFD objects can be used and combined. You cannot obtain a useful analysis from DFDs that violate these rules, so please study them carefully.

First, the names of all data flows touching the same process must differ; it is illogical to have input and output data flows of the same name, since this indicates that the process does not transform the data that are flowing through it. The one exception to this rule is mentioned as part of the following rule.

Second, data flows can be shown connecting processes with other processes, stores, or terminators, but data flows cannot be shown connecting a store with a store, a store with a terminator, or a terminator with a terminator. This is because a data flow directly between stores or between a store and a terminator provides no way to move the data between the two (whereas a process does), and a data flow between terminators has nothing to do with the task being analyzed. If the nature of a task requires that data be moved between a store and a store or a store and a ter-

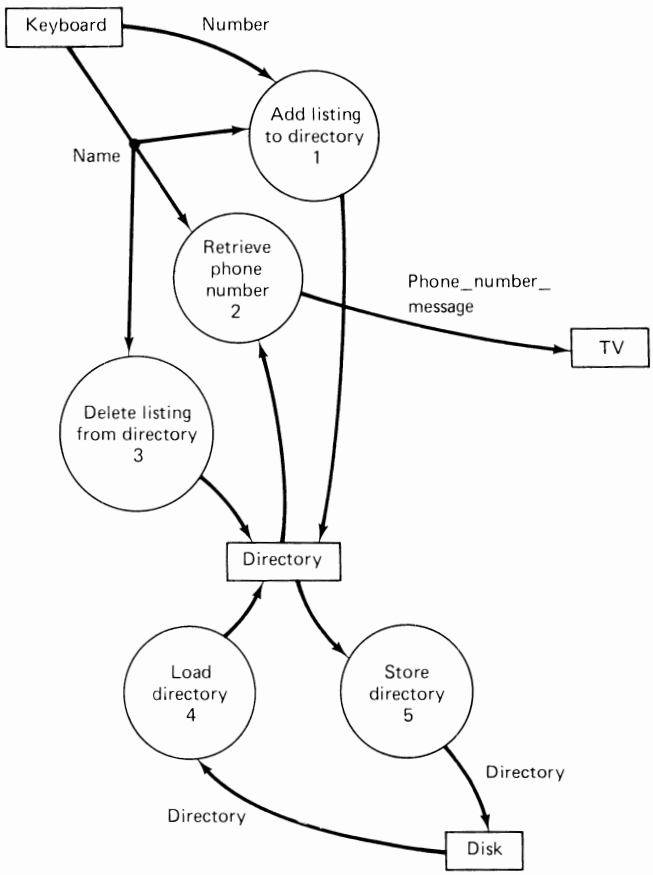


Figure 4.4

minator, it is acceptable to violate the first rule and show a process that moves the data without transforming it. This is the exception to the first rule that we noted.

Third, every process must be connected to both input and output data flows. Clearly, processes that do nothing with something (i.e., have only input flows) or that produce something from nothing (i.e., have only output flows) contribute nothing to our understanding of a task that produces something from something. They are also impossible to implement in programs.

Fourth, each data item should be shown entering a process in only one place. This removes a source of redundancy and therefore of confusion. For instance, in the manage_directory task you would not show 'name' and 'listing' entering the same process, because 'listing' contains 'name', and violates this rule. If these two data elements were used by the same process, you would probably show 'name' and 'number' input data flows under the assumption that the process would put them together to form a listing.

Fifth, trivial error data can be shown on a DFD as a short data flow from a

process into midair. Thus we could have shown short "no such listing" arrows exiting the "retrieve phone number" and "delete listing from directory" processes.

Sixth and last, as we have already said, information flows used to control or coordinate the execution of processes should not be shown on a DFD. However, status flows can be shown if they are essential to performing the task.

We mentioned that a task image consists of a *set* of data flow diagrams. We will discuss why more than one DFD is necessary, and how the set is developed, when we study the analysis process.

We can now discuss the second item in a task image: the data dictionary.

Data dictionary. The data dictionary of the task image holds definitions of the data flows and data stores. These definitions are in the format used for design-model definitions. Indeed, the data dictionary developed during the analysis phase is used as the starting point for the design-phase data dictionary. Low-level data definitions such as 'number = 10{(ASCII DIGIT)}10' are omitted from the dictionary during the analysis phase because, as we said, analysis defines the inherent parts of a task without considering how they will be implemented. For an example of a data dictionary, see the data dictionary section of the earlier discussion of the design phase.

The final item in a task image is the collection of the process specifications for the processes making up the overall task.

Process specifications. The process specifications consist of pseudocode definitions of what the various processes making up the overall task do. They take on the same form as the module specifications of the design phase. The only difference between process specifications and module specifications is that process specifications show no control or coordination decisions, whereas module specifications describe such implementation-dependent actions. For an example of the format for process specifications see the module specifications section of the earlier discussion of the design phase.

The Analysis Process

Analyzing a task is a two-stage process. First, the scope of the task must be determined. Then the task must be divided into its component processes and their data interfaces.

Determining task scope. The scope of a task is determined by carefully defining what information the task must exchange with which objects in the outside world, and what time constraints, if any, apply to the task's completion.

The task's interfaces with the outside world are determined with a special type of DFD called a *context diagram*. A context diagram shows the entire task as a single process circle, with the terminators around the task and the interface data flows connecting the task and the terminators.

Example:

Show the context diagram for the manage_hierarchy task.

The context diagram in Fig. 4.5 shows the single high-level process 'manage_hierarchy', surrounded by the data sources and sinks with which it must communicate. Notice that it is consistent with the earlier DFD example, but with the internal details of the task missing.

Drawing the context diagram is the most important step in structured analysis, for it defines the precise limits on the scope and nature of a task. Your first step in drawing a context diagram is to identify the terminators, or data input and output devices. With a task for a personal computer the terminators are typically things like TV sets, disk drives, data cassettes, and keyboards. Draw the identified terminators, with appropriate names, around an empty circle that represents the task. You will name the task later.

Second, define the data flows outward from the circle to the terminators. These are the data produced by performing the overall task. They are the desired products of executing that task. The output data flows must be identified first because you must know exactly what results you want from performing a task before you can determine what input data or internal processes are required to achieve those results. The data definitions of the output data flows should be the first entries in your data dictionary. In the manage_directory task we might have the definitions:

```
directory = 1{listing}100
listing = name + number
phone number message = (*phone number in displayable form*)
```

The first of these definitions differs from its counterpart in the design model in that it omits the 'directory_size' data element. This happens because we are not yet con-

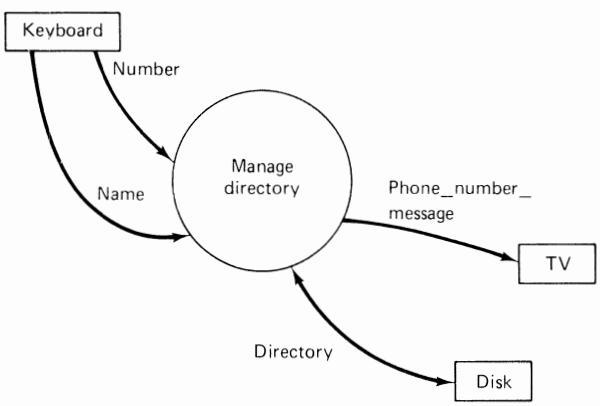


Figure 4.5

sidering such design issues as "How will a module know the size of the directory data structure?" We would add that information later, when we become aware that such an action is necessary. Like all other analysis and design tools, the data dictionary need not be in a perfect and final form on the first try; it only needs to be consistent within itself.

Third, examine those output data flows and determine the minimal input data from which the outputs can be derived. Each data item should be input in only one place. This is easier to do than it may seem. For instance, in the manage_directory task we know that a directory and phone number messages are output. It is clear that phone numbers must first be input to later be output, and it is also clear that names must also be input to define listings fully and to select particular phone numbers. So 'name' and 'number' are the only input flows necessary. Note in the preceding paragraph that the data dictionary definitions for the output flows told us that 'name' and 'number' would be needed. If we decide that we want to be able to reload an existing directory from the disk drive, 'directory' will also be an input data flow, although strictly speaking, it is not absolutely necessary for the function of the task. Even with more complicated tasks than 'manage_directory', the input data necessary to produce the output data are usually easily determined. If they are not, you should identify all the necessary input data that you can and correct any errors in your choices later as they are by the analysis process. You should also record definitions for the input data flows in the data dictionary.

Fourth and finally, look over the data interfaces and try to understand from them the exact nature of the task. Then choose the most applicable, specific, and "strong" (i.e., imperative) name for the overall task. Write that name inside the task circle. This step has been delayed until now because, for the reasons noted under the first step in this sequence, it is the data interfaces that determine the nature of the task. One wants the task name to reflect the task nature as closely as possible.

At this point the context diagram is complete. However, with some tasks the execution time from receipt of the data inputs until production of the data results is a critical factor. This consideration can also be shown on a context diagram. One of the better ways of doing this is to draw a dashed line between each time-critical output data flow and the input flows used to produce it, and to note the time limit for the transformation beside the dashed line. You would refer to this information during the programming phase to select algorithms and programming strategies that meet the noted time limits.

Example:

Show a time limit of 0.5 seconds from the time a name is entered from the keyboard until the corresponding phone number is displayed on the TV screen.

Figure 4.6 (see pg. 190) shows how this is done.

Dividing the task. After the context diagram has defined the scope of a task, the processes needed to perform that task must be identified together with any

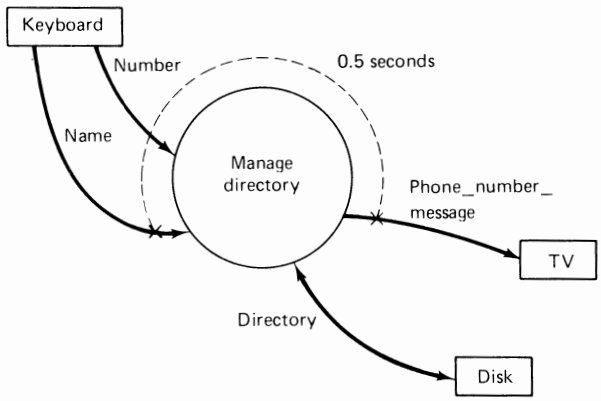


Figure 4.6

necessary data interfaces between them. We do this with the techniques of structured analysis.

The Techniques of Structure Analysis. A task is divided into its component processes using two techniques; *top-down leveling,* which identifies the component processes with increasingly detailed levels of DFDs, and *bottom-up leveling,* which is usually used to improve the quality of existing analyses.

Top-Down Leveling. The levels of DFDs we will use to divide a task begin with the most basic DFD, the context diagram, which shows a task being performed by a single all-encompassing process. Using top-down leveling, we will separate out the many little subprocesses in the overall process. However, those subprocesses are often too numerous to show at the same time in a single DFD. When that is the case, we use a DFD to divide the overall process into between three and nine intermediate subprocesses, more DFDs to divide each of those subprocesses into between three and nine more, and so on until we have DFDs that show the most basic subprocesses and their data interfaces. The DFDs together form a leveled set that describe in various levels of detail the processes needed to perform a task.

The overall process shown in the context diagram is divided from the outside in, starting with the external output and input data interfaces. To work inward along an output data flow you must imagine what data a last-step subprocess would require to produce the output data. You need not identify or name the last-step subprocess yet; you only need to name the data that subprocess uses to produce the output.

When dividing a complex task, the data required to produce the output will be intermediate in content and structure complexity to the external task inputs and outputs. In simpler tasks the identified flows might be the task inputs.

After you have identified the data needed to produce the output flow, draw a smaller circle inside the overall task circle and connect it to the new data flows and to the output data flow. It helps to start with a large copy of the context diagram so

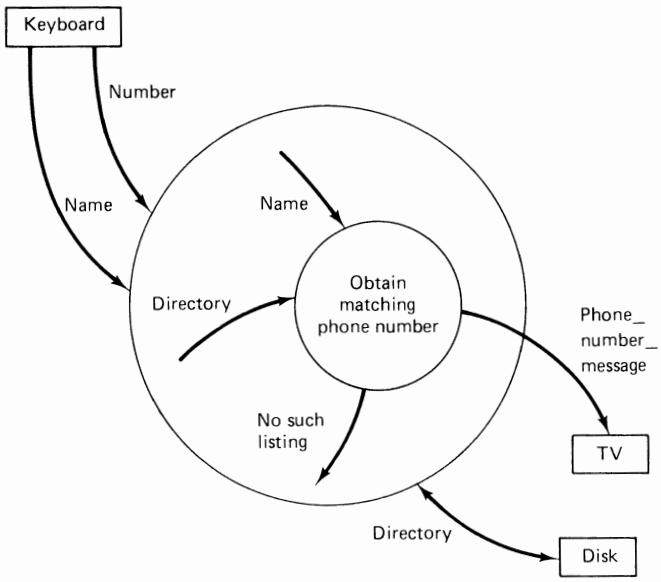


Figure 4.7

that there will be plenty of room for what you will draw inside the process circle. A large black or white board is an ideal place to make this copy, since it will allow you to easily make the many corrections that will be needed as you go along.

Once you have drawn the inner subprocess circle, give it a strong process name based on its surrounding data interfaces, in the same way that you named the overall task after its external data interfaces.

Example:

Show how the subprocess producing the 'phone-number message' might be identified.

The 'phone-number message' can be produced directly from a 'name' data flow and the 'directory' data structure. The 'manage_directory' context diagram with the added data flows and added subprocess circle is shown in Fig. 4.7.

The data interfaces to the 'obtain matching phone number' subprocess inspired its name. Note that this process corresponds to the 'retrieve phone number' process in the 'manage_directory' DFD at the end of the DFD section. Choosing the process name based on its data interfaces has made it stronger and more descriptive than before. Note also that we have not yet connected the 'directory' data flow into this process with the task input flow of the same name. This is because we know that a store will be needed to hold the directory. Thus we can work inward one more step and show the store now (Fig. 4.8).

Dividing a task along an input interface is done in a similar way. Imagine what data flows can be produced from the interface data in a logical first-step process. Such a process will often require the data from more than one input interface. Once the necessary input interfaces and produced data have been identified, draw another

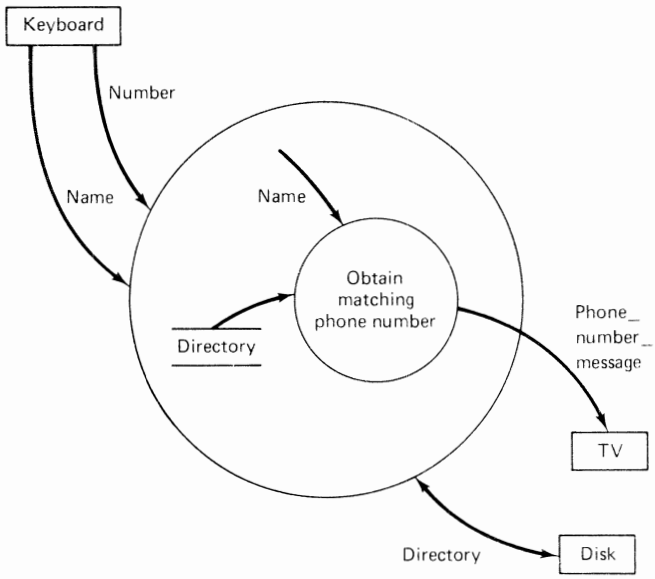


Figure 4.8

small circle inside the overall task circle and connect it with the identified data flows. Then give the new circle a strong process name based on its data interfaces. As with dividing along output interfaces, depending on the complexity of the task, the newly identified data flows may or may not be external interfaces.

Example:

Show how the subprocess that first uses the 'directory' input might be identified.

The 'directory' store and 'directory' input interface have already been defined. The DFD rules say that a process must be used to move data from a terminator to a store. Thus we connect the 'directory' input flow to a subprocess circle and connect the circle with a data flow to the store. The result is shown in Fig. 4.9.

The dangling data flows left on the interior of the task circle by previous inward divisions are treated like external interfaces and worked further inward to identify other data flows and transforming processes. The goal is to work the inputs and the outputs toward each other until they are connected by transforming processes. When a complete data path exists from the external inputs to the external outputs, the level 0 DFD is complete. The top-level process circle is no longer shown in the level 0 DFD.

To divide a task in this way you must take the passive viewpoint of observing what happens to the data, rather than the active viewpoint of deciding what job must be done next. You are observing objects rather than deciding upon actions. This makes task division with DFDs a fundamentally different kind of activity than, say, drawing a flowchart as we did in structured programming. A passive, observing

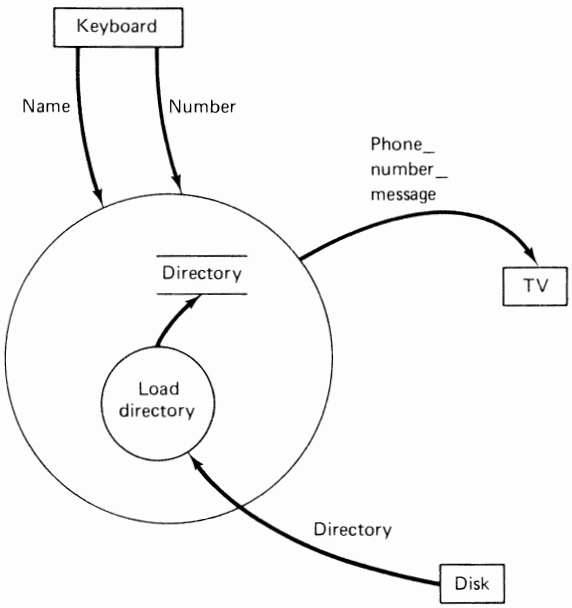


Figure 4.9

method is better suited to large tasks, while an active, decisive method is better with very small tasks. Each has its place.

Example:

Show the complete level 0 DFD for the manage_directory task.

This DFD combines the process divisions of the last two examples with other similar divisions until the task has been completely divided into five processes and one data store. The DFD appears as shown in Fig. 4.10. Note that this is the same DFD as was shown in the DFD section example except for the name change of the 'obtain matching phone number' process.

Each process in the level 0 DFD can be treated like the overall process in the context diagram and divided in a separate DFD by working inward from its surrounding interfaces. A DFD used to divide a process shown in the level 0 DFD is given the number of the process being divided. So a DFD showing the division of the 'add listing to directory' process would be called the level 2 DFD, and would have as its external interfaces the data flows 'name', 'number', and 'directory'. The processes in this level 2 DFD would be numbered 2.1, 2.2, and so on. A DFD dividing the first process in the level 2 DFD would be the level 2.1 DFD. This division continues until DFDs are obtained whose processes cannot be divided any further without weakening their cohesiveness, where "cohesion" has the same implications of unity of purpose and function that it had with modules.

With leveled DFDs, no single DFD divides an entire task, but the context

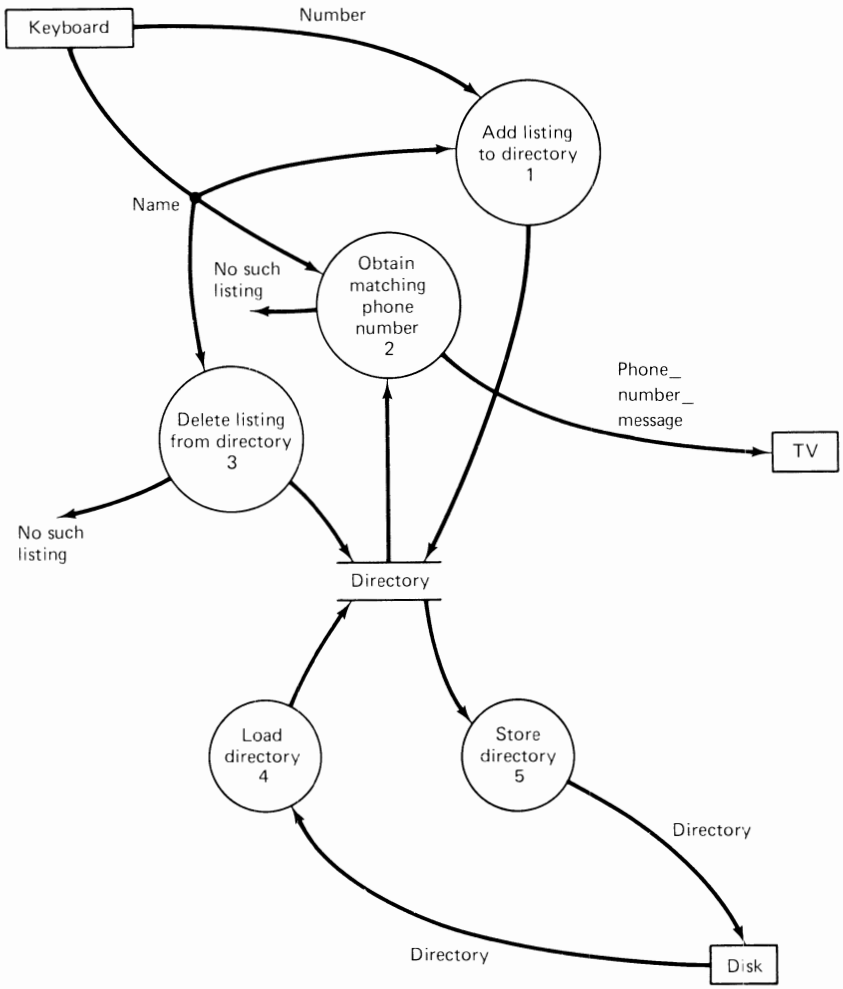


Figure 4.10

diagram and level 0 DFD together provide a task overview. With simple tasks such as 'manage_directory' these two DFDs may even identify the basic processes. Otherwise, still lower-level DFDs will reveal the basic processes required to perform the task.

All the DFD rules we mentioned earlier apply to leveled DFDs. Additionally, the data flows into and out of a process in a DFD must match the data flows into and out of the DFD that expands that process. This ensures conservation of data, and that the lower-level DFD accurately describes what must be done to perform the overall process it describes.

When the most basic processes have been identified, their functions can be

defined in pseudocode process specifications. These are exactly like the module specifications used during design; indeed, the module specifications will be developed from the process specifications. As we noted earlier, process specifications omit all control or coordination decisions (e.g., 'when status_flag = TRUE THEN do action ELSE wait'). Process specifications should state only what the process will do with input data that we assume are always available and ready for processing.

Example:

Finish leveling the manage_directory task.

Only two processes in the level 0 DFD can be divided without weakening their cohesion. They are 'obtain matching phone number', process 2, and 'delete listing from directory', process 3. Even these processes are so simple that their divisions are trivial. Thus the level 2 and level 3 DFDs are all that are needed to complete the division of the manage_directory task. The level 2 DFD is shown in Fig. 4.11.

The level 3 DFD is similar. Try drawing it to practice what we have discussed.

Leveled DFDs can also be used to divide noncomputer tasks; indeed, DFDs in one form or another have been used for analyzing tasks since long before the electronic computer was invented. For instance, data flow diagrams were and are used by contractors to divide the task 'build a house' into the smaller processes such as 'build a foundation' that make up the task. For this type of task the data flows are replaced by material flows consisting of the wood, concrete, pipes, cabinets, and so on, that are used in the various construction processes. Each process is checked for conservation of material, rather than conservation of data. The basic principles are otherwise the same.

DFDs have also been used to analyze the main task performed in an office. In this case the material flows are the office paperwork, and dividing the task in the best possible way leads to fewer steps between desks and therefore saves time and money.

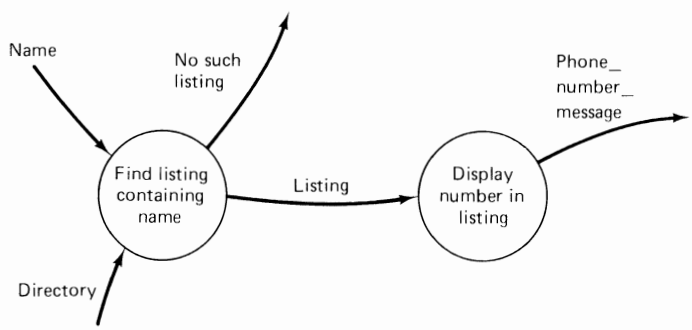


Figure 4.11

Finally, the processes in such jobs are defined with pseudocode process specifications. These descriptions are actually procedures for performing each process.

With a team of workers, such as are found on construction sites and in offices, parallel processing is the norm. You might have bricklayers putting up a fireplace while finishers are putting in the sheetrock, and so on, and similarly in an office.

Thus DFDs are useful for understanding tasks of many sorts, and can be used profitably to gain control over your noncomputer projects as well as over your programs.

Bottom-Up Leveling. The second technique of structured analysis, bottom-up leveling, is used to correct weaknesses in an existing leveled analysis. It is usually used when too many data interfaces connect the processes shown on intermediate-level DFDs, although it can also correct other analysis weaknesses. Thus bottom-up leveling is primarily used to decrease coupling.

Loose coupling between modules was a prime goal in the design phase. Loose coupling between processes is just as important during analysis. Coupling as it applies to DFDs is discussed in the upcoming laws of structured analysis section, so you might read that discussion before going any further.

The idea behind bottom-up leveling is quite simple. Normally, the basic processes that perform a task are shown in a group of bottom-level DFDs. The next-higher-level DFDs group these processes together, DFDs another level higher group the groups, and so on, until all processes have been grouped together into the single process of the context diagram. This is what the DFD levels represent, although they are developed in the opposite way, by dividing downward rather than grouping upward. If the analyst has divided the processes in less than the best possible way, there will be more interfaces between the intermediate-level processes than necessary. To correct this, the analyst reverses the process and groups processes upward starting with the most basic level.

The procedure for working upward is as follows. Start with a blank black or white board, or with a large piece of paper (e.g., 4 feet by 4 feet), and copy onto it all the lowest-level processes from the bottom-level DFDs. Copy their data connections so that the entire system is shown, from the external inputs through all the basic processes to the external outputs.

Now, look at the arrows and circles, and circle the groupings that result in the fewest arrows between the circles. Each grouping should contain between three and nine circles or low-level processes. You might want to draw the grouping circles in a different color from the rest of the chart so that they will stand out.

By choosing the groupings based on least coupling, you have identified the strongest processes to represent on the next-to-lowest level. You will find that these groupings are easier to name than the previous processes on that level, and the names should be stronger than before. If you cannot choose strong names for the new groupings, try regrouping at the same level for the same or even fewer interfaces

between groups. You should find that the most loosely coupled groupings are easiest to name.

Now, draw a new chart with just the groupings and their interfaces. Group and name them as before, and you will have the second level up from the bottom. Continue this process until you have worked back up to the context diagram, and the improved set of leveled DFDs is complete.

Note that in bottom-up leveling you do not consider data flow names or process names, only the number of arrows between groupings. This keeps the use of the upward-leveling technique simple and quick.

The work you did in top-down leveling is never wasted, because until you know what the basic, indivisible processes at the most detailed level of a task are, you cannot improve the analysis. At the very least you will be keeping the top and bottom levels of your original analysis; the bottom-up leveling process then allows you to improve the intermediate levels. These levels will be particularly important when we convert the analysis results into an initial design, as you shall see.

Example:

Show how bottom-up leveling can improve the coupling of an intermediate-level DFD.

Since bottom-up leveling is done based on arrows and circles, not on data or process names, we will label the data flows with numbers and the processes with letters. This will ensure that we think only in terms of fewest interfaces and do not become entangled in the words accompanying the figures. We use uppercase letters for the intermediate-level DFD and lowercase letters for the low-level DFD.

The intermediate-level DFD shows tight coupling between its processes and a need for correction from below by upward leveling, as we see in Fig. 4.12. Processes A and C have been downward-leveled into bottom-level DFDs, which are shown below. Process B is indivisible and cannot be leveled further. The low-level DFD expanding process A appears as shown in Fig. 4.13.

Note that the processes making up A have no data connections between them. As we said in the discussion of structured design, bad coupling and bad cohesion go hand in hand. The processes in A have terrible cohesion; they do not have *anything* in common!

The low-level DFD expanding process C appears as shown in Fig. 4.14. We now plug in the low-level processes of A and C into the original intermediate-level DFD, to form a composite low-level DFD (Fig. 4.15).

Next we will regroup these low-level processes to form new intermediate-level processes with the fewest connecting arrows possible (Fig. 4.16). We will call these new groupings X and Y.

Last, we draw the new intermediate-level DFD with just the two new intermediate-process circles (Fig. 4.17). Where before we had three processes with two, four, and five interfaces, respectively, now we have two processes with just two and three interfaces, respectively. We could also form just one process with only three interfaces if that seems more appropriate to the underlying system.

A variation on bottom-up leveling can be used in place of top-down leveling during the entire analysis phase for relatively simple tasks. Start as before with a

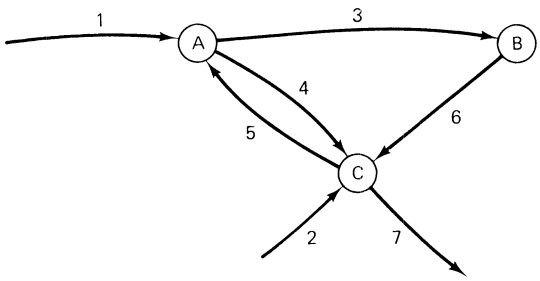


Figure 4.12

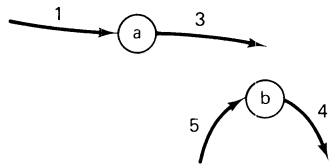


Figure 4.13

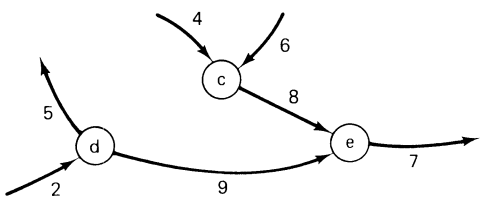


Figure 4.14

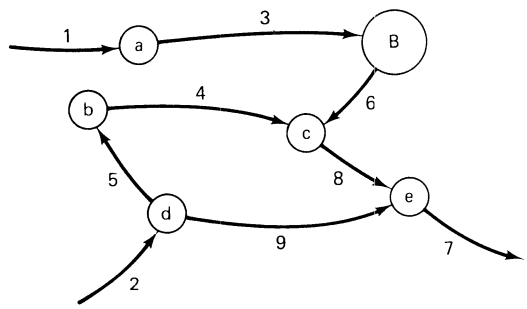


Figure 4.15

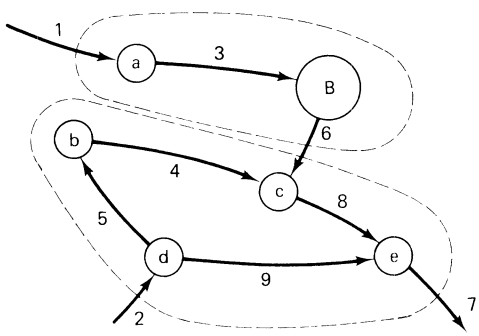


Figure 4.16

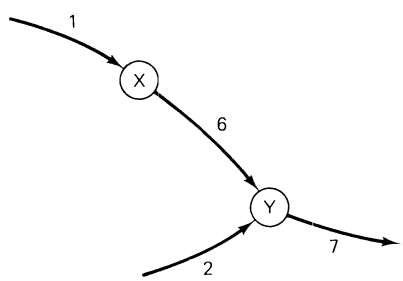


Figure 4.17

context diagram and a large surface to write on. Then work inward, identifying and notating the lowest-level processes you can. Often it is easier to identify basic processes than it is to select appropriate intermediate-level processes. When you have finished with this step, you will be ready to level upward until reaching the context diagram. The result is the same as with top-down leveling; a set of leveled DFDs describing the processes required to perform a task. Of course, this method cannot be used if the task is so large that its basic processes cannot be written on a single chart.

Bottom-up leveling of an existing set of DFDs can be used with a large task if you pregroup the basic processes that are most tightly coupled. Multiple bottom-level charts will be required, but at least the basic processes that are most likely to be functionally related will be on the same charts. These basic charts are leveled upward until the earliest time that a single chart can be made from the process groupings. The upward leveling then continues as usual through the context diagram.

Bottom-up leveling corrects problems identified by the law of coupling. As in the design phase, we use laws to identify problem areas in our work. The remaining laws of analysis point out other improvements needed in the results of an analysis. We discuss the laws of analysis next.

The Laws of Structured Analysis. Many of the laws of structured analysis have design counterparts with which you are already familiar. In those cases we will highlight the differences in how the law is applied during analysis, and move on. The other laws are relatively simple and intuitive, and will be easily grasped and applied. The laws are discussed in their general order of importance.

Conservation of Data. This law is the cornerstone of structured analysis. It is defined just as in structured design, except that at this stage it is processes that must exhibit a connection between all inputs and outputs, instead of modules.

The law states that "all process input data must be used to produce process output data, and all process output data must have corresponding process input data from which it is derived." Again, it means that data cannot be created from nothing, nor can they disappear into thin air.

This law is usually violated in one of two ways. First, an interface to a process in a given DFD can accidentally be left out of the lower-level DFD used to divide that process. Second, the context diagram may have too few or too many external interfaces. In the first case, the cure is to redraw the lower-level DFD with the omitted interface and to continue leveling from there. In the second case, the cure is to start the analysis over. Even then, all is not lost. Whatever work you have done before the error was discovered will improve your feeling for the task on the next try.

The law of conservation of data is the only check for the completeness of solution of an analysis, so the latter law is not needed.

Nontemporality. The law of nontemporality states that "temporal considerations must be omitted from the task image." Temporal considerations include

control and coordination information flows, which control when the processes execute what, and control-related decision making in the process specifications.

Common violators of this law include processes that act based on the time element, whether it be execution time or real time, and control flows produced by such processes to activate and deactivate other processes. For instance, given the task of maintaining an executive's appointment book, its DFDs might show a time-aware process to 'output appointment message on target date'. This process might activate a 'display appointment message' process, so that both the first process and its control flow violate the law of nontemporality.

Violations of this law reveal that the person making the analysis has been thinking actively, of action sequence, rather than passively observing the path the data take through the system. This has limited the direction of the analysis to conform to his or her preconceived ideas of what the system should do. At best, a program based on such an analysis may work but not as well as it otherwise would have. At worst, a program based on such an analysis will be too complicated to get it working properly, if at all.

Violations of this law can often be corrected simply by removing the offending processes and control flows. However, if this leaves dangling data flows, the cure is to return to the level above the first introduction of temporal elements and level downward without them from there.

Coupling. This law is defined similarly to its relative in structured design. It states that "process interfaces should be as few and as nonintrusive as possible."

As we said in the discussion of bottom-up leveling, violations of this law reveal an imperfect division of the task into its processes. The problem is corrected with bottom-up leveling.

Cohesion. The law of cohesion as it applies to analysis is similar to the law of cohesion for structured design. It states that "each process should consist only of directly related functions, and as much as possible, all directly related functions should be grouped together in the same processes."

Cohesion in a task image is most easily tested with the law of coupling and by the strength of the process and data-flow names. Vague or grab-bag process names indicate a lack of relatedness between process functions. Vague or grab-bag names for data flows or stores indicate that they consist of loosely related or bundled data structures and that the process producing them probably has weak cohesion.

Processes exhibit the same types of cohesion as modules. In both cases only functional, sequential, and communicational cohesion are acceptable. Bottom-up leveling is used to correct weak cohesion.

Balance. The law of balance for structured analysis says that "the data inputs to any process in a DFD must be of similar complexity to the outputs." If this law is followed, each process in a DFD can be divided into its basic component processes in about the same number of levels as any other process in the same DFD.

Lack of balance reveals an uneven division of processes during leveling, which if carried to extremes leads to complications in structured design and programming. Imbalance is corrected with bottom-up leveling.

Process Size. The law of process size states that "the specifications for the basic processes identified by top-down leveling should require one half page of pseudocode or less." Of course, the figure of one half page is somewhat arbitrary, but it is a useful yardstick nevertheless.

Violations of this law indicate that the most basic processes have probably not yet been identified. This makes it much more difficult to write and read the program code that will eventually perform those processes. A long process specification often implies less than optimal cohesion, as well. Further top-down leveling of the offending processes corrects this problem.

Summary of Structured Analysis

Structured analysis uses two techniques and six laws to produce a task image in the form of a set of leveled DFDs, a data dictionary, and process specifications.

The two techniques of structured analysis are:

1. *Top-down leveling:* dividing the overall task process shown in a context diagram into the most basic processes required to perform that task.
2. *Bottom-up leveling:* regrouping the basic task processes to produce intermediate levels of loosely cohesive processes.

The six laws of structured analysis are:

1. *Conservation of data.* All process input data must be used to produce process output data, and all process output data must have corresponding process input data from which it is derived.
2. *Nontemporality.* Temporal considerations must be omitted from the task image. Temporal considerations include control and coordination information flows, which control when the processes execute what, and control-related decision making in the process specifications.
3. *Coupling.* Process interfaces should be as few and as nonintrusive as possible.
4. *Cohesion.* Each process should consist only of directly related functions, and as much as possible, all directly related functions should be grouped together in the same processes.
5. *Balance.* The data inputs to any process in a DFD must be of similar complexity to the outputs.
6. *Process size.* The specifications for the basic processes identified by top-down leveling should require one half page of pseudocode or less.

From Analysis to Design

The analysis phase is complete with the finishing of a task image that obeys all the laws of structured analysis. The design phase begins at this point with the conversion of the task image into an initial design model. We will make the conversion using the transform analysis technique.

Transform analysis. Transform analysis creates the leveled hierarchy directly from the leveled set of DFDs in the task image. The conversion is so straightforward that it could almost be automated.

Transform analysis starts with the level 0 DFD. The first step is to collapse any data store together with its interface data flows into the main process that writes to it. In the phone directory problem, this would require hiding 'directory' within the 'load directory' process. Remove any trivial error flows.

The DFD now shows only major data flows, data transformers, and terminators. The second step is to remove the terminators from the DFD, leaving the external interfaces dangling.

Example:

Prepare the level 0 manage_directory DFD for conversion into a hierarchy chart.

After hiding the 'directory' store, removing the 'no such listing' error flows, and removing the terminators, we have the result shown in Fig. 4.18.

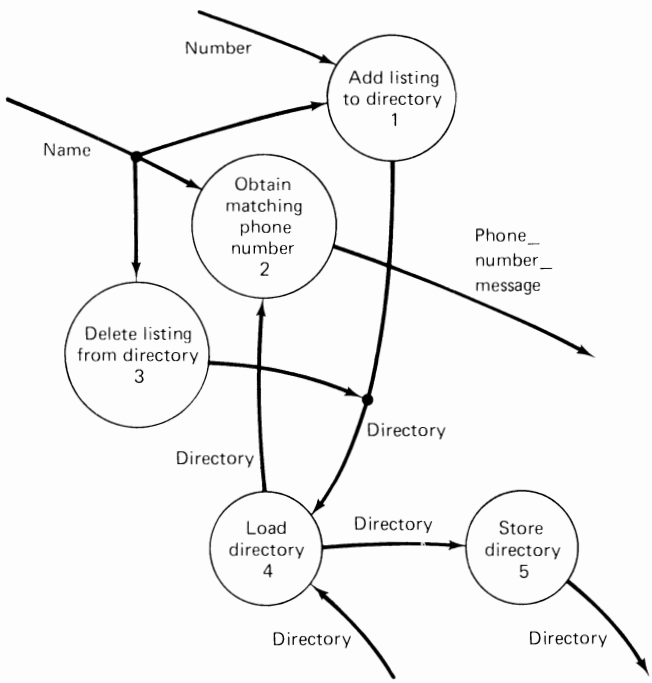


Figure 4.18

Once the level 0 DFD has been prepared, you must find the central process or processes of the DFD. This process or group of processes will perform the central function of the overall program and will determine the name of the top or 'boss' module in the design hierarchy. These central function processes are also called the *central transform,* for obvious reasons.

There are two ways to find the central transform. The first is to search the DFD for the processes that seem to be most involved in the central data transformations of the task. Such processes tend to be more highly coupled with each other than with the surrounding data input- or output-related processes, although this is not always true. Looking for a more tightly coupled group of data-transforming processes can often lead you to the central transform.

The second way of finding the central transform uses the process of elimination instead of the direct approach. You trace all external input and output data flows inward until reaching the most logical (i.e., the most general or highest-level data structure) form of the data. If you make a mark across each of these data flows and then connect them to form a circle, all processes inside the circle will be involved in the transformation of those data flows and therefore will be in the central transform. This method is usually better than the first, since solid principles rather than intuition lead us to the central transform.

Example:

Apply the second method of finding the central transform to the prepared manage_directory DFD.

Working inward along the external data flows leads us to the central transform shown circled in Fig. 4.19.

Once you have the central transform circled, imagine that the DFD processes are ping-pong balls and that the connecting data flows are string. If you pick up all the balls in the central transform in your hand, the other balls will hang underneath by their threads. Note that the result is beginning to look like a hierarchy chart.

Cut the threads that cross the circle of the central transform, and glue the cut ends to a new ping-pong ball, which we will call the 'boss' ball. Give the boss ball the imperative name that best describes the function of the central transform.

If the central transform contains just one process, hang it directly beneath the 'boss' as well. If the central transform contains more than one process, you can hang each of them under a 'supervisor' ball and then hang the 'supervisor' under the 'boss', or you can hang each of them directly under the 'boss'. Closely related processes in the central transform favor the first approach, but since the rest of the design phase will be spent perfecting the initial hierarchy, either approach is acceptable.

At this point you may also want to remove the names of the data flows to or from any data stores, and consider all the connected processes as having equal but unshown access to them. Data stores tend to act as common couples between pro-

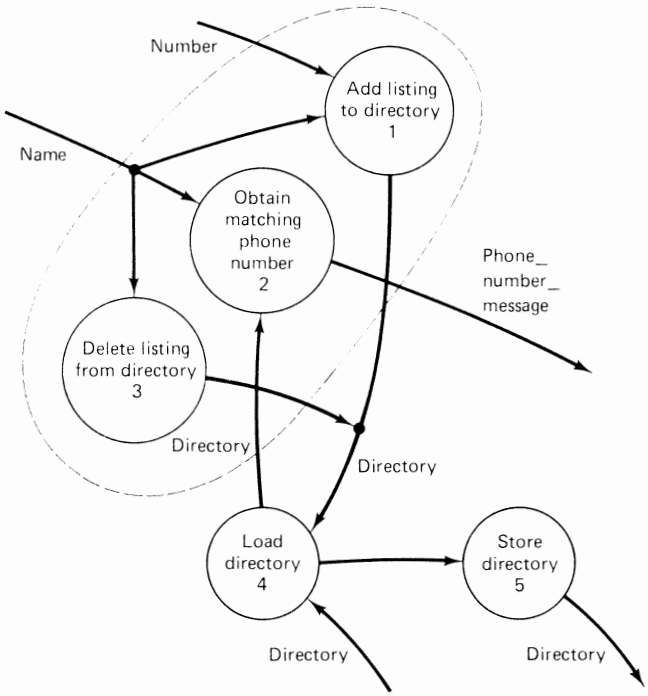


Figure 4.19

cesses, and showing the data interfaces through the 'boss' process clutters up the diagram with many copies of the same data flow. In programs that revolve around data stores, the process names will usually refer to the store already, and any accessing of the stores will show up in the process pseudocode.

Example:

Show the manage_directory DFD after it has been placed in ping-pong form.

Since the three processes in the central transform are only loosely related, we will give the 'boss' ball the general but weak name 'manage_directory', and hang each of the processes directly from the 'boss' (Fig. 4.20). We will not show the 'directory' flows since they flow to and from a data store.

To complete a ping-pong chart, look at the DFDs expanding each process now hanging below the 'boss'. Now treat each hanging process as a 'boss' in its own right, and hang the lower-level DFD beneath it using the same procedure as above. Do this level by level until the lowest-level DFDs have been hung on the chart.

To complete the format of the initial hierarchy chart, replace the process circles with module rectangles, add the data interface arrows beside the data flow names, and hang an I/O module on each dangling-end external interface. The latter type of module formats data as necessary to meet the differing needs of the external

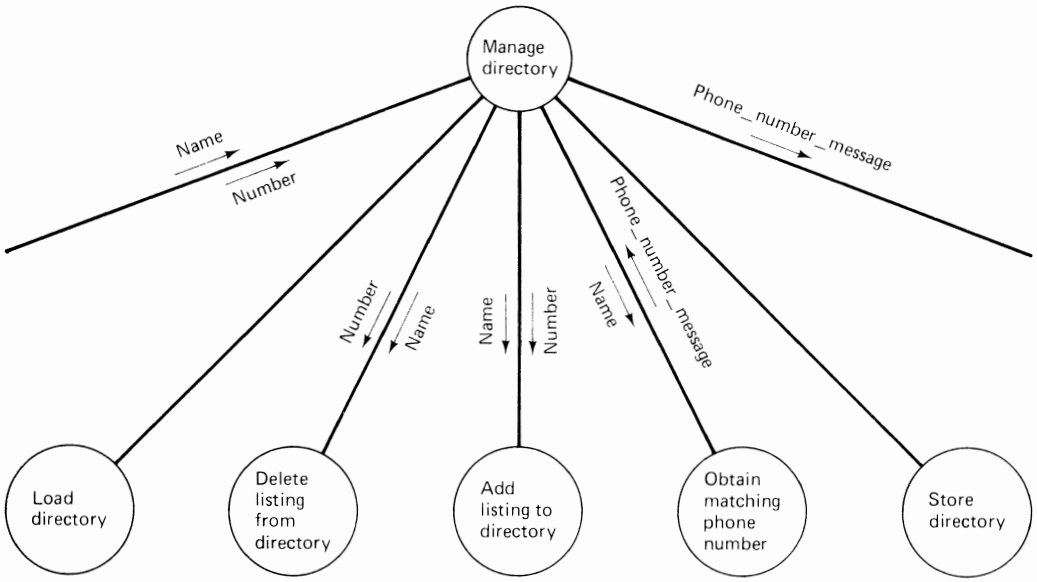


Figure 4.20

device and the computer, and handles any protocol needed to input or output that data. For an input flow from a keyboard such a module might provide prompts and canned selections to guide the user in providing the correct data. For an output flow to a TV screen such a module might place the data in labeled columns and add remarks on their significance. Of course, only one module is needed on any one type of dangling end, no matter how many low-level DFDs the interface appears in.

There are no dangling interfaces in our ping-pong manage__directory chart, but there would have been if we had not kept 'directory' in a data store. In that case, the directory would have been loaded directly from the disk terminator each time it was used by a process, and stored to disk whenever it was updated. The 'load__directory' and 'store__directory' processes would have been omitted in the original DFD since they do not transform the directory. In creating the ping-pong chart, the removal of the terminators would leave the directory interfaces dangling. When converting the ping-pong chart into a module hierarchy we would have added 'load__directory' and 'store__directory' modules at the end of these loose ends to move the data from and to the disk.

To complete the initial design model, add any necessary control interfaces to the hierarchy chart and data dictionary and any necessary control logic to the pseudocode specifications of the modules. Completing the 'manage__directory' initial design model is left to you for practice. There is no one right answer at the end of a transform analysis process; it is absolutely necessary only that you include all processes from the DFDs, and all data flows except those to and from data stores, if you

choose to omit those. Almost any other flaw can be corrected during the rest of the design phase, including oversights in adding the I/O and control aspects needed to make the hierarchy work in the real world. Do your best with identifying the I/O and control needs, however, since this can save you a lot of extra work later. The I/O and control aspects must also be correct by the end of the design phase.

The completion of the initial design model starts the second portion of the structured design phase, perfecting the model. This brings us full circle to the structured design section earlier in this chapter and completes our discussion of the software development process.

Exercise:

Work through the following problems to practice the software development process.

(a) Analyze, design, and write your own program for a telephone directory. Refer to Chapters 3 and 4 as necessary for help with the techniques, but do not try to duplicate the system we described exactly. Add enhancements to vary the project from the examples in this chapter, but use the examples as a reference to guide you through the development.

(b) Develop a program from one of the following categories or a simple category of your own choosing (you will need information from the next three chapters to program the I/O modules, but the analysis and design phases can be completed with what you know now).

Four-function calculator (+, −, ×, /)
Home inventory
Tic-tac-toe
"Memo pad" word processor (*Hint:* Data structures are the key.)

FOR FURTHER STUDY

Subject	Book	Publisher
Structured programming	*Top-Down Structured Programming Techniques* by C. L. McGowan and J. R. Kelly	Petrocelli-Charter
Structured design	*The Practical Guide to Structured Systems Design* by Meiler Page-Jones	Yourdon Press
Structured analysis	*Structured Analysis and System Specification* by Tom DeMarco	Yourdon Press

CONNECTING THE NERVES: USING THE MEMORY MAP

Everything you do with your computer involves the memory map, or address space, of the 6510 CPU. Data and programs are stored in the memory map, and all communications with the outside world are made through locations in the memory map. We are now ready to discuss how you use the locations in the Commodore 64 memory map to accomplish your programming goals.

ADDRESSABLE LOCATIONS

As we noted in Chapter 1, the memory map of the 6510 CPU is 64K bytes long. However, the Commodore 64 provides a total of 88K addressable locations from which to select the 64K, including 20K of ROM, 64K of RAM, and 4K of I/O data and control locations. These locations can be divided and combined in a number of different ways so that the memory map can be tailored for specific tasks. We will start by exploring the ROM, RAM, and I/O locations, and then see how they are combined to form the different memory maps.

ROM Locations

The Commodore 64 is supplied with one built-in data structure and two built-in programs. These are kept in permanent form in 20K bytes of ROM. The built-in data structure is the 4K-byte-long *character set*. The built-in programs are the BASIC language interpreter and the operating system. These programs are 8K bytes long each, and are discussed below.

Character set ROM. The character set data define the graphical appearance of the characters shown on the Commodore 64 keyboard. This data structure will be described in greater detail in Chapter 6. It can optionally be in or out of the CPU memory map. In either case it is still available to the graphics chip, which accesses it independently of the CPU as we will also see in Chapter 6. When the character set ROM is in the memory map, it is always located between addresses D000 and DFFF hex.

BASIC interpreter ROM. The BASIC interpreter is a built-in program that takes a user-written BASIC program that has been loaded into RAM and converts it into CPU-executable machine language instructions. The interpreter converts one user-program instruction at a time and feeds the resulting machine language instructions to the microprocessor for execution. Thus the microprocessor, under the control of the interpreter, alternates executing the interpreter and executing machine language instructions that perform a single instruction in the user's BASIC program. This process is illustrated in Fig. 5.1.

The BASIC ROM can also be in or out of the CPU memory map. When it is in

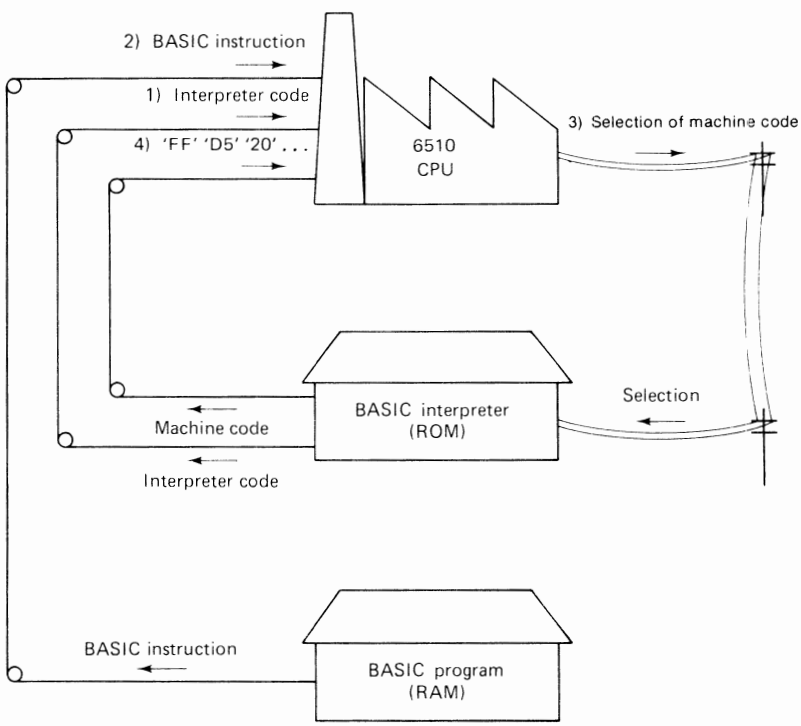


Figure 5.1 Interpreting a BASIC instruction (CPU inputs and outputs are numbered by order of use)

the map it is located at addresses A000 through BFFF hex. The computer does not need the BASIC ROM to run machine language programs except at the outset to load a program and start it executing. Once executing, the machine language program can select a memory map that replaces the BASIC ROM with RAM locations in its address range.

Kernel ROM. The operating system, also called the *kernel,* is more a collection of machine language modules than a program, although some kernel modules call other kernel modules to perform their tasks. However, the modules together do perform one broad overall task; supporting I/O communications with the outside world using the Commodore 64's I/O circuits. This task often involves complicated housekeeping functions requiring specialized programming knowledge not needed for anything else. The kernel simplifies our programming job by providing easily used modules that our programs can call to perform those functions. Figure 5.2 shows how a program uses a kernel module to perform I/O functions.

When a machine language program calls a kernel module, it passes it any necessary data in the 6510 registers. Any data produced by executing the module are returned in the registers and are then handled by the program.

A module in a typical program hierarchy calls another module with a JSR to the beginning address of the called module. We could use the same method to call kernel modules if we had a list of their beginning addresses. Such a list could be made easily enough. However, Commodore has stated that the beginning addresses of the kernel modules may be changed as module features and length are altered. Such changes could be due to changes in the I/O chips, for instance, which might require a change in the housekeeping duties and therefore length of the kernel modules. More likely, extra capabilities could be added to or hidden errors corrected in the modules. A table of module addresses would become obsolete as soon as one module address changed, and so would the programs built using the old table.

For a program to be sure to run on every Commodore 64, including your own, it must call the kernel modules without using their beginning addresses. A clever trick allows this to be done.

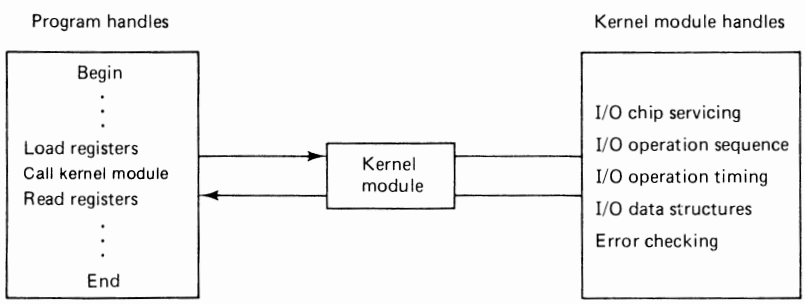


Figure 5.2

Inside the kernel ROM of every Commodore 64, starting at address FF81 hex when kernel ROM is in the memory map, is an ordered group of 39 JMP instructions. These instructions point to 39 kernel modules also in kernel ROM, and are in a known order so that the JMP instruction for any particular kernel module is at the same address in every Commodore 64. The group of JMPs is called the *kernel jump table.*

To call a particular kernel module, a program loads the CPU registers with any necessary data and calls the location holding the JMP instruction for the desired module. Execution passes to the JMP instruction, which routes execution to the beginning address of the module in that particular C64. When the module finishes its job, execution returns to the location following the original call. Any data from the kernel are then retrieved from the CPU registers. We see how this works in Fig. 5.3.

The complete JMP table is shown in Table 5.1, with the destination module of each JMP named and its purpose summarized. You will use only about half of the JMPs and kernel modules during normal programming; most of the remaining modules are directly called by this basic group. The JMPs you will use in your programs are highlighted in the chart. The modules they point to form the nucleus of the Commodore 64's operating system; they perform most input and output with a simple technique called *channel I/O.* Understanding channel I/O is the key to using the operating system, so we will return to this subject later in the chapter.

So, according to Table 5.1, the kernel module called CHKOUT can be called by executing the program instruction JSR $FFC9. This causes the JMP instruction at $FFC9 to execute, which routes execution to the module CHKOUT. If you are curious where CHKOUT or other modules are located in your Commodore 64, you can find their addresses by using a machine language monitor to look at the last two locations of each three-byte JMP instruction. Absolute addressed JMPs contain the module address in the last two bytes of the instruction, with the low byte of the ad-

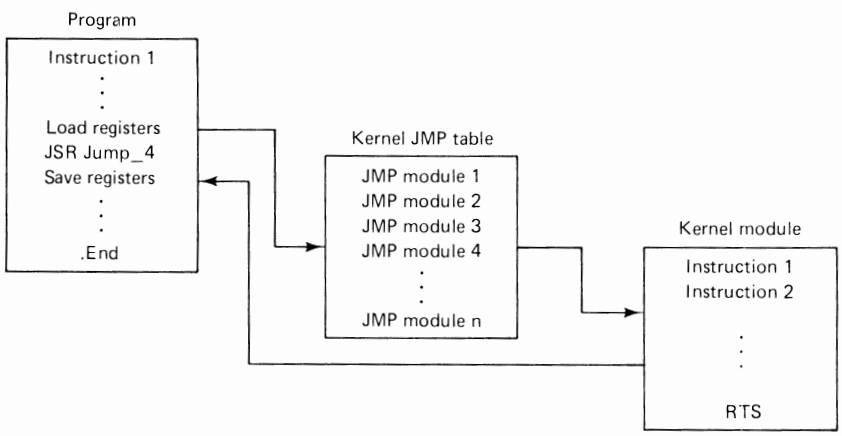


Figure 5.3

TABLE 5.1 DETAILED ROM MEMORY MAP: KERNEL JMP TABLE

JMP	Module	Purpose
FF81	CINT	Initialize video chip and screen editor
FF84	IOINIT	Initialize I/O chips and I/O module variables
FF87	RAMTAS	Initialize low RAM to BASIC's requirements
FF8A	RESTOR	Initialize kernel pointers
FF8D	VECTOR	Move kernel pointers
FF90	**SETMSG**	**Select type of message from kernel**
FF93	SECOND	Send secondary address command to listener
FF96	TKSA	Send secondary address command to talker
FF99	MEMTOP	Read or set top of BASIC's program memory
FF9C	MEMBOT	Read or set bottom of BASIC's program memory
FF9F	**SCNKEY**	**Put pressed key into keyboard queue**
FFA2	SETTMO	Set/reset device wait flag for IEEE add-on card
FFA5	ACPTR	Load the accumulator from serial bus device, with handshaking
FFA8	CIOUT	Send accumulator byte to serial bus device, with handshaking
FFAB	UNTLK	Make a serial bus talker a nontalker
FFAE	UNLSN	Make a serial bus listener a nonlistener
FFB1	LISTEN	Make a serial bus device a listener
FFB4	TALK	Make a serial bus device a talker
FFB7	**READST**	**Determine status of I/O transfer**
FFBA	**SETLFS**	**Define a communications link**
FFBD	**SETNAM**	**Name a communications link**
FFC0	**OPEN**	**Add a link to the open table**
FFC3	**CLOSE**	**Terminate link and close file**
FFC6	**CHKIN**	**Select a link as input channel**
FFC9	**CHKOUT**	**Select a link as output channel**
FFCC	**CLRCHN**	**De-select channels, set to keyboard and screen channels**
FFCF	**CHRIN**	**Wait for and get a byte from the input channel**
FFD2	**CHROUT**	**Send a byte to the output channel**
FFD5	**LOAD**	**Open channel, load PRG file, close channel**
FFD8	**SAVE**	**Open channel, save PRG file, close channel**
FFDB	SETTIM	Set the kernel clock
FFDE	RDTIM	Read the kernel clock
FFE1	**STOP**	**Check for stop key**
FFE4	**GETIN**	**Get a byte from keyboard queue or RS-232**
FFE7	CLALL	Terminate all links and channels without closing files
FFEA	UDTIM	Increment kernel clock
FFED	SCREEN	Get row/column organization of video screen
FFF0	**PLOT**	**Get row/column cursor position**
FFF3	IOBASE	Get beginning address of I/O block

dress first. Indirect addressed JMPs contain the address of the memory locations holding the module address.

With a monitor program you can also write short *test modules* that call kernel modules and watch them execute one step at a time. There is no better way to learn how an operating system works, but it is a challenging and sometimes tedious business.

The Commodore VIC 20 has a jump table that is nearly identical to that in the Commodore 64, and Commodore claims intent to do the same with future compatible computers. Thus Commodore 64 programs using the kernel jump table should translate easily to other Commodore computers, as long as the program is compatible in other ways (e.g., in program size and use of the I/O chips).

As with the BASIC ROM, the kernel can be in or out of the CPU memory map. When the kernel is in the map it is always located at addresses E000 through FFFF hex.

RAM Locations

The 64K bytes of RAM locations are used to hold user programs and their data as well as any temporary data handled by the two built-in ROM programs. RAM is located at addresses 0 through FFFF hex.

The first four 100 hex pages of RAM are used by the BASIC interpreter and the kernel to store the temporary data they produce as they execute. The BASIC locations are available to most machine language programs since such programs normally remove the BASIC interpreter ROM from the memory map. Additionally, locations used by any kernel modules or functions that the program never uses will also be available. This releases some page-zero locations for your use, which allows using the faster page-zero addressing. However, be sure not to use any locations assigned to kernel modules that will be used, since dual use leads to errors in both the kernel and your program.

Data used by the VIC graphics chip to produce the computer's output picture are also stored in RAM. These data are kept in one of several predefined data structures, depending on the type of display. Most graphics data can be placed in a number of different areas in RAM. However, a special-purpose graphics data structure called Color RAM can only be placed in addresses D800 to DFFF hex. We discuss graphics data in detail in Chapter 6.

All areas of RAM not used by the built-in programs or for graphics data are available to hold your programs and their data. During programming you will have to select a beginning address for the program to rest in memory. Under certain circumstances that we will discuss shortly (e.g., auto-load), you may also have to specify beginning addresses for modules located apart from the main program. In selecting memory locations for your program the only rule is to make sure that the entire program rests in otherwise unused RAM locations. This means that the locations you have available in which to place your program will depend on the specific memory map that you select.

The reserved low-address RAM locations are shown in more detail in Tables 5.2 and 5.3. Table 5.2 is a summary of the divisions of low memory into BASIC, kernel, and graphics sections. Table 5.3 lists the specific kernel and BASIC data locations that are most important to assembly language programming. Do not expect the contents of these charts to be very meaningful your first time through. Later

TABLE 5.2 SUMMARY LOW-RAM MEMORY MAP

Address	Used by	Purpose
0000	I/O	6510 data direction register
0001	I/O	6510 I/O register
0002-0003	Program	Beginning of area used by ROM code
0003-008F	BASIC	
0090-00FA	Kernel	Variables and pointers
00FB-00FE	Program	
00FF-010A	BASIC	
0100-01FF	Stack	
0200-02FF	Kernel	Variables and pointers
0300-030B	BASIC	Pointer at 0302-0303 used for auto-start
030C-03FF	Kernel	Variables and pointers
0400-07FF	Graphics	Default screen memory data structure
0800	Program	Beginning of main user program area

they will be useful as a programming reference, but for now they will have served their purpose if they introduce you to the major uses of low RAM.

Each section of low RAM is shown with its address range in hexadecimal notation, what type of program uses it (PROGR means its free for use by user programs), and in some cases, what it is used for.

So the locations in address pages 0 and 2 through 3 store temporary data for kernel modules, for the memory-map and other configurations (through the 6510's I/O and data direction registers at addresses 0 and 1), for the BASIC interpreter, and for your programs if you desire. Since most I/O operations performed by your programs will use the kernel modules, the page 0 kernel locations should be left alone. All the BASIC locations are free for use by your assembly language programs. The two BASIC locations that are shown in the following detailed RAM map are those that you must use in designing programs that load and start executing automatically under the control of the BASIC interpreter. We will see later how these locations are used.

Page 1 locations, at addresses 0100 through 01FF hex, make up the stack area,

TABLE 5.3 INDIVIDUAL LOW-RAM LOCATIONS

Address	Name	Purpose (main accessing modules)
00A0-00A2	TIME	Kernel clock
0259-0262	LAT	Open table: link #s (open)
0263-026C	FAT	Open table: device #s (open)
026D-0276	SAT	Open table: commands (open)
0277-0280	KEYD	Keyboard queue (SCNKEY, GETIN)
02A1-0301	ENABL	Cassette: free for disk auto-start loader
0302-0303	IMAIN	Basic pointer used to start auto-start loader
0314-0315	CINV	Pointer to IRQ interrupt routine
0316-0317	CBINV	Pointer to BRK interrupt routine
0318-0319	NMINV	Pointer to NMI interrupt routine

as discussed in Chapter 1. Locations in pages 4 through 7 (addresses 0400 through 07FF hex) and between addresses D800 and DBFF hex make up the default screen and color graphics areas, respectively. These areas are selected by the kernel and loaded with data when the computer is first turned on. They can be freed for program use by reassigning the graphics data to other address areas. Again, we will study these data structures and their usage in Chapter 6.

Table 5.3 shows individual low-RAM locations of two types. Locations of the first type will be directly accessed by your programs, while locations of the second type are accessed only by the kernel but are shown for reference to help you understand the kernel when we discuss it. Each location is listed with its address in hexadecimal and decimal notation, its name as found in Commodore's literature (to help you cross-reference their manuals), a description of its contents, and the names of the major kernel modules that write to or read from it in parentheses.

I/O Locations

The I/O locations are clustered in a 4K-byte block that can be in or out of the memory map. When present in the memory map this block is always located at addresses D000 through DFFF hex.

Some of the I/O locations serve as the windows to and from the outside world via devices such as printers, cassette drives, disk drives, and game joysticks. There are also built-in devices connecting the computer to the outside world, including timers and an alarm clock. Another group of I/O locations are used to control the windows. The remaining I/O locations are unused.

Each of the I/O locations is a register on one of the four I/O chips in the Commodore 64. One chip supports graphics, one supports audio, and two support most other I/O operations. The graphics chip is called the VIC-II, which is short for Video Interface Controller model II. The audio chip is called SID, for Sound Interface Device. Finally, the general-purpose chips are called CIAs, for Complex Interface Adapters. In this chapter we introduce the first two chips and then concentrate on the latter two. The following chapters fill in the details on the graphics and audio chips.

The kernel utilizes the I/O locations to carry out its I/O functions. We will use kernel modules for I/O wherever possible. However, some I/O functions are not supported by the kernel, and for these functions we must use the I/O locations directly. Locations of both types are shown in Table 5.4, with the addresses they reside at when the I/O block is in the memory map.

THE MEMORY MAP

From the 88K bytes of available addressable locations, a 64K memory map must be selected and electrically attached to the microprocessor. 64K bytes of RAM are always at the foundation of the memory map, brokenly or continuously spanning

TABLE 5.4 DETAILED I/O-BLOCK MEMORY MAP

Address	Name	Purpose
D000–D02E	**VIC-II**	**Graphics-chip area of memory map**
D000–D010	Sprite positions	X and Y coordinates of sprite
D000		Sprite 0 X coordinate (LSByte)
D001		Sprite 0 Y coordinate (LSByte)
D002		Sprite 1 X coordinate (LSByte)
D003		Sprite 1 Y coordinate (LSByte)
D004		Sprite 2 X coordinate (LSByte)
D005		Sprite 2 Y coordinate (LSByte)
D006		Sprite 3 X coordinate (LSByte)
D007		Sprite 3 Y coordinate (LSByte)
D008		Sprite 4 X coordinate (LSByte)
D009		Sprite 4 Y coordinate (LSByte)
D00A		Sprite 5 X coordinate (LSByte)
D00B		Sprite 5 Y coordinate (LSByte)
D00C		Sprite 6 X coordinate (LSByte)
D00D		Sprite 6 Y coordinate (LSByte)
D00E		Sprite 7 X coordinate (LSByte)
D00F		Sprite 7 Y coordinate (LSByte)
D010		X coordinate MSBit, all sprites
D011	Cntrl rgstr A	Bits 0–2: Vertical pixel scroll Bit 3: Select 24/25 row display Bit 4: Blank screen (0) Bit 5: Mode (1 = bit map, 0 = not bit map) Bit 6: Submode (1 = xback, 0 = not xback) Bit 7: MSbit of raster register (D012)
D012	Raster regstr	Raster position; raster IRQ latch
D013	Light pen X	Pen horizontal position (halved)
D014	Light pen Y	Pen vertical position
D015	Sprite enable	1-Bit enables corresponding sprite
D016	Cntrl rgstr B	Bits 0–2: Horizontal dot scroll Bit 3: Select 38/40 column display Bit 4: Submode (1 = mcol, 0 = char) Bit 5: *Must equal* 0! Bits 6,7: Not used
D017	Vertcl expand	1-Bit enlarges corresponding sprite
D018	VIC offsets	Bits 1–3: Char-set offset in VIC 16K map Bits 4–7: Screen offset in VIC 16K map
D019	IRQ flags	1 = interrupt has occurred Bit 0: Raster position Bit 1: Sprite/background collision Bit 2: Sprite/sprite collision Bit 3: Light pen triggered Bit 7: Any of the above
D01A	IRQ enable	1-Bit enables corresponding D019 IRQ
D01B	Dsply priorty	1-Bit sprites display over background
D01C	Sprite submod	1-Bit sprites are multicolor mode
D01D	Hrzntl expand	1-Bit enlarges corresponding sprite
D01E	S/S collision	1-Bit = sprite collided

(*continued*)

TABLE 5.4 *(Continued)*

Address	Name	Purpose
D01F	S/B collision	1-Bit = sprite collided
D020	Border color	
D021–D024	Backgnd color	
D021		Color 0
D022		Color 1
D023		Color 2
D024		Color 3
D025–D026	Multicolor rgs	
D025		Multicolor register 0
D026		Multicolor register 1
D027–D02E	Sprite color	
D027		Sprite 0 color
D028		Sprite 1 color
D029		Sprite 2 color
D02A		Sprite 3 color
D02B		Sprite 4 color
D02C		Sprite 5 color
D02D		Sprite 6 color
D02E		Sprite 7 color
D02F–D3FF	—	(unused)
D400–D41C	**SID**	**Audio chip area of memory map**
		All regs are write-only except that D419–D41C are read-only
D400–D406	Voice 1 cntrl	
D400	Frequency	Low byte
D401	Frequency	High byte
D402	Pulse width	Low byte
D403	Pulse width	Bits 0–3: High nibble, bits 4–7 unused
D404	Control rgstr	Bit 0: Start ADS (1), start R (0) Bit 1: Sync freq with voice-3 freq Bit 2: Modulate freq with voice-3 freq Bit 3: Voice switch (1 = OFF, 0 = ON) Bit 4: Triangle waveform (1 = ON, 0 = OFF) Bit 5: Ramp waveform (1 = ON, 0 = OFF) Bit 6: Pulse waveform (1 = ON, 0 = OFF) Bit 7: Noise waveform (1 = ON, 0 = OFF)
D405	AD envelope	Bits 0–3: Decay time (0 to F) Bits 4–7: Attack time (0 to F)
D406	SR envelope	Bits 0–3: Release volume (0 to F) Bits 4–7: Sustain time (0 to F)
D407–D40D	Voice 2 cntrl	Same As D400–D406 except for the Control register at D40B: Bit 1 syncs freq with voice 1 (1) Bit 2 mods freq with voice 1 (1)
D40E–D414	Voice 3 cntrl	Same As D400–D406 except for the Control register at D412: Bit 1 syncs voice 3 with voice 2 (1) Bit 2 mods voice 3 with voice 2 (1)

TABLE 5.4 *(Continued)*

Address	Name	Purpose
D415–D418	Filter cntrl	
D415	Cutoff freq	Bits 0–2: Low-order frequency bits
D416	Cutoff freq	High byte
D417	Filter Switch	Bit 0: Filter voice 1 (1 = yes, 0 = no)
		Bit 1: Filter voice 2 (1 = yes, 0 = no)
		Bit 2: Filter voice 3 (1 = yes, 0 = no)
		Bit 3: Filter external input (1 = yes)
	Fltr resonance	Bits 4–7: Filter resonance (0 to F)
D418	Output volume	Bits 0–3: From 0 to F
	Filter mode	Bit 4: Lowpass mode (1)
		Bit 5: Bandpass mode (1)
		Bit 6: Highpass mode (1)
		Bit 7: Turn voice 3 audio off (1)
D419–D41A	Game pdl regs	D419: L paddle, D41A: R paddle
D41B	Oscillator 3	Waveform value for voice 3 (nmbr gen)
D41C	Envelope 3	ADSR value for voice 3
D41D–D7FF	—	(Unused or undependable)
D800–DBFF	—	(Area left free for color RAM underneath)
DC00–DCFF	**CIA #1**	**Complex I/O support chip**
DC00	Data port A	Bits 0–7: Kernel output to query keybd
		OR
		Bits 0–3: Port 2 joystick input OR
		Bits 2–3: Port 2 L and R paddle firing button inputs (0 = pressed)
		Bit 4: Port 2 joystick firing button input (0 = pressed)
		Bit 6: Paddle port 1-select output (1)
		Bit 7: Paddle port 2-select output (1)
DC01	Data port B	Bits 0–7: Input from keyboard read
		OR
		Bits 0–3: Port 1 joystick input OR
		Bits 2–3: Port 1 L and R paddle firing button inputs (0 = pressed)
		Bit 4: Port 1 joystick firing button input (0 = pressed)
DC02	DDRA	Data direction register for port A
		1 sets bit to output, 0 to input
DC03	DDRB	Data direction register for port B
DC04–DC07	Timer bytes	Used by kernel; see timer regs CIA #2
DC08–DC0B	Clock/alarm	Clock output or alarm input (see DC0F)
DC08		1/10-seconds byte
DC09		Seconds byte
DC0A		Minutes byte
DC0B		Hours byte, bit 7 = AM/PM (PM = 1)
DC0C	Serial I/O	Used by the kernel
DC0D	IRQ flags	Bit 2: Clock alarm (1)
		Bit 7: IRQ occurred (1)

(continued)

TABLE 5.4 *(Continued)*

Address	Name	Purpose
DC0E	Cntrl rgstr A	Bit 0: 0 = stop timer cycle, 1 = start
DC0F	Cntrl rgstr B	Bits 0–6: Do not change Bit 7: Clock/alarm bytes: 0 = clock output
DC10–DCFF	—	(Unused)
DD00–DD0F	**CIA #2**	**Complex I/O support chip** Only free (non-kernel) registers are shown
DD00	Data port A	Bits 0–1: Select VIC 16K data area 00 = C000–FFFF, 01 = 8000–BFFF, 10 = 4000–7FFF, 11 = 0000–3FFF Bits 2–7: Do not change
DD02		Bits 0,1: 1 = enable DD00 area select
DD04–DD07	Timer bytes	
DD04		Timer A low byte
DD05		Timer A high byte
DD06		Timer B low byte
DD07		Timer B high byte
DD0D	IRQ flags	Bit 0: Timer A interrupt (1) Bit 1: Timer B interrupt (1) Bit 7: IRQ occurred (1)
DD0E	Cntrl rgstr A	Bit 0: Start/stop Timer A (1 = start) Bits 1–2: *Always reset to* 0's Bit 3: Timer A mode; 0 = continuous, 1 = one-shot Bit 4: Load timer latch into timer counter (1) Bits 5–7: Do not change
DD0F	Cntrl rgstr B	Bits 0–4: Same as cntrl rgstr A applied to timer B Bit 5: Timer B mode; 0 = count 1-MHz pulses, 1 = count timer A empties Bit 6: *Always reset to* 0
DD10–DFFF	—	(Unused)

addresses 0 to FFFF hex. Variety in the memory maps comes from placing ROM over specified areas of RAM, and from replacing one specific area of RAM with the I/O circuits block.

Placing ROMs over the foundation RAM is different from replacing RAM altogether: a microprocessor reading from a ROM location that is *over* RAM retrieves the ROM contents as usual, but one writing to such a location puts data into the RAM locations underneath the ROM. So, although the microprocessor sees the currently selected memory map, the video chip has access to all of RAM, although only 16K at a time (the functioning of the video chip is described in Chapter 6). One can update video display data that are stored in RAM underneath

ROM in a current memory map, and thereby change the TV picture drawn by the video chip.

When in the memory map, the BASIC ROM lies over the RAM from A000 to BFFF hex. Similarly, the kernel ROM lies over the RAM from E000 to FFFF hex, and the character ROM over the RAM in the shared RAM-I/O area from D000 to DFFF hex.

Most of the I/O block replaces rather than overlays RAM. This means that the RAM is completely disconnected from the address bus so that the CPU cannot write through the I/O locations to it. This is necessary because the I/O chips have read/write locations like RAM, so that writing to I/O over RAM would change both locations identically. However, the color RAM between addresses D800 and DBFF hex holds important data about the color of the TV display that must be program changeable. Therefore, this address range in the I/O block has been left vacant and data can be written through the I/O block into the color RAM beneath.

Although most of the RAM in the I/O block address range is detached from the CPU when the I/O block is present, these RAM locations are always available to the VIC chip, which has its own memory map that never includes the I/O block.

The various CPU memory maps are selected by writing binary values into the three least significant bits of the 6510's on-board I/O register, at address 0001 hex (Fig. 5.4).

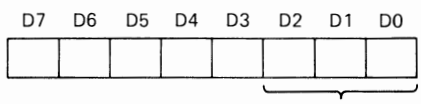


Figure 5.4 I/O Register (0001 hex)

Usage of the upper 5 bits is divided between cassette control and no defined function. These higher-order bits are usually handled only by the kernel routines, but their contents must still be preserved when we change the lower 3 bits. The safest way to do this is to retrieve the byte in location 0001 hex to the accumulator, AND or OR the accumulator with bit patterns that change only the desired bits, and return the accumulator's contents to location 0001 hex.

With three memory map bits, up to eight (2^3) different memory maps can be selected. However, two of the maps are identical, reducing the total number of unique maps to seven.

Overview Map

As the name "memory map" implies, the 64K address space can be shown pictorially. In the overview memory map shown in Fig. 5.5, all 88K available locations are pictured at their assigned addresses. The individual memory maps are con-

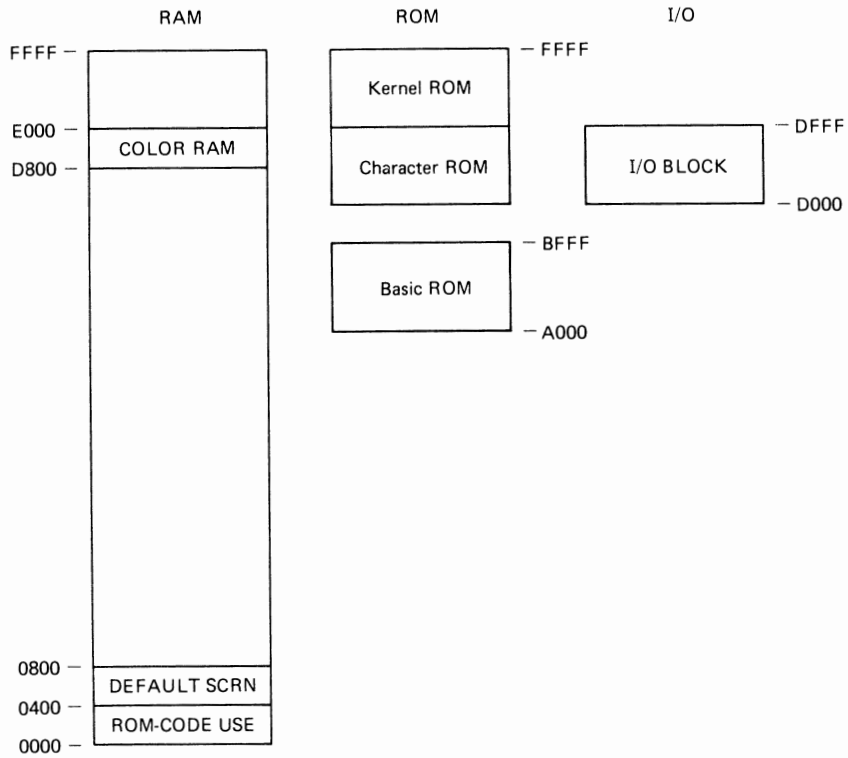


Figure 5.5 Overview memory map

structed by selecting locations of one type in each area where two or more types of locations are allowable.

In Fig. 5.5, the 64K RAM foundation map appears at left, labeled with addresses. The ROM and I/O areas that overlay or replace portions of it are shown to the right beside the locations they occupy.

Selecting the map. The three ROMs and the I/O block are attached to and detached from the memory map using the memory map bits in the 6510's I/O register at address 0001 hex. A table of the bit patterns for attaching these sections (Table 5.5) resembles the binary counting table. In this table, all possible values of bits d0, d1, and d2 of the I/O register are shown together with the contents of their resulting memory maps. The table starts with d2 equaling 1 instead of 0 as in the normal counting sequence, to show the most important memory maps first. Also, memory maps that are especially, or perhaps only, well suited to a single purpose are labeled as such (e.g., "assembly" for holding assembly language programs).

Note that changing d2 from a 1 to a 0 merely substitutes the character ROM for the I/O block, except in the all-RAM maps. The main reason for making this substitution is to make the character set available for transfer into RAM and subse-

TABLE 5.5 MAP OPTIONS

Use	(0001 hex) D2:D1:D0	Map contents
ASSEMBLY	1 0 0	64K RAM
	1 0 1	60K RAM, 4K I/O
	1 1 0	52K RAM, 4K I/O, 8K kernel
BASIC	1 1 1	46K RAM, 4K I/O, 8K kernel, 8K BASIC
DATA TRANSFERS	0 0 0	64K RAM (SAME AS 100)
	0 0 1	60K RAM, 4K character set
	0 1 0	52K RAM, 4K character set, 8K kernel
	0 1 1	46K RAM, 4K character set, 8K kernel, 8K BASIC

quent alteration. To transfer this data structure successfully, interrupts must be disabled with a SEI instruction beforehand, and reenabled with a CLI instruction afterward. The infrequent need to do this is what makes the first four maps the most useful for normal programming.

Two bits, provided by lines in the expansion port connector behind the Commodore 64, select additional memory maps. However, the programmer cannot use them without building a cartridge to plug into the port. The extra bits allow a cartridge to insert its own ROM into the main memory map. The few people who might be using these bits will probably already own a copy of Commodore's *Programmer's Reference Guide,* which includes their memory maps and electrical diagrams of the ports and the Commodore 64. The electrical diagrams are particularly useful for this kind of computer work, so anyone interested in the subject should buy the reference guide. Since the extra maps are of no use to most assembly language programmers, we will not discuss them further here.

The four major memory maps can be interchanged as a program executes for various purposes. For instance, a program may start out under the BASIC map, switch to the 64K RAM assembly language map and load much of memory with graphics data, and then alternate between the three assembly language maps as necessary to access all of RAM and still use the I/O block and the kernel ROM.

Our next step is to learn to use the memory map locations we have just surveyed. In this chapter we will be studying the kernel, one aspect of the BASIC interpreter, and several aspects of the I/O block. We will study the character set ROM in Chapter 6, the remaining aspects of the I/O block in Chapters 6 and 7, and RAM requires no further comment. We start with the kernel, which communicates with I/O devices via a technique called *channel I/O.*

USING THE KERNEL: CHANNEL I/O

In channel I/O, a program uses the kernel to send and receive data through a communications channel between the CPU and a nonmemory device. The concept can be illustrated with a historical example. Prior to the presidency of John F. Kennedy,

there was no means for the President of the United States and the Soviet head of state to communicate quickly and reliably. Of course, there were communications paths such as radio and telephone between the two countries, but in their normal form none of these were sufficient for the special needs during tense situations.

In 1963, existing communications paths were tailored and combined with communications devices called teletypes at each leader's end to form a communications link called the "hot line." The hot line is inactive except in crisis situations between the two countries. When activated, however, it becomes a communications channel for carrying information directly between the national leaders.

The Commodore 64 also has links and channels. A link consists of a communications path connected to the computer on one end and to an I/O device or peripheral on the other. As with the hot line, a Commodore 64 link is normally in an inactive state. To communicate with a peripheral a program must both create a link and activate it using the kernel. Thus a *channel* is an activated link.

Channels are unidirectional; that is, a given channel can input or output data, but not do both. Two channels are supported by the kernel: one for input and one for output. However, there is a rarely used method that circumvents the kernel to allow multiple output channels. We will discuss it later.

When the computer is first turned on, or when channels created by a program have been dissolved, the kernel forms the input channel from the keyboard and its communications path, and the output channel from the TV screen and its communications path. These are the *default channels.* A program can replace these channels with channels for communicating with any other peripheral attached to the Commodore 64. When a program finishes using its selected channels and deactivates them, the kernel automatically reselects the default channels.

The kernel provides several ways to move data through a channel. Depending on which communications path and device have been selected, data may be moved as a complete file data structure, as parts of a file, or as sentence or character units tailored for human input or human viewing. We will call these choices *whole-file I/O, partial-file I/O,* and *interactive I/O,* respectively. This flexibility allows a program to transfer only the amount of data needed for a given purpose. The computer is unavailable for program execution while data are being transferred, and some of the communications paths are excruciatingly slow (such as those to the disk or cassette drives), so the ability to transfer a little data at a time and get back to other work can be invaluable.

The steps in using a channel are quite simple. First, a program creates a communications link containing a peripheral and the communications path to which it is attached. Second, the program activates the link as an input or output channel. Third, the program communicates with the peripheral via the channel. Fourth, when communications have been completed, the channel is deactivated. Finally, the link is dissolved. These steps are summarized below:

1. Create a link.
2. Activate the link.

3. Communicate via the channel.
4. Deactivate the channel.
5. Dissolve the link.

How these steps are executed depends on the type of channel being used. Thus, to understand how to use a channel from a program, we must first be familiar with the component parts of channels: communications paths and peripheral devices. These are discussed next.

Channel Parts

Peripherals are designed to work with specific communications paths, so we will discuss both the paths and the devices in terms of the paths.

Communications paths. For a computer to be useful it must communicate with the outside world. Over the years computer-equipment manufacturers have developed communications paths for connecting computers and peripherals. Communications paths are defined by rules governing the allowable number of electrical lines, voltages, size and shape of connectors, coordination of signals, and other similar characteristics.

A few of these paths have prospered and become *standards*. Having standard communications paths has allowed manufacturers to build peripherals that work with the products of other manufacturers. Users benefit by having more varied and less expensive devices to attach to their computers.

The two most common path standards are nondescriptively named IEEE-488 and RS-232. The Commodore 64 partially supports each of these standards, allowing the use of peripherals such as disk drives, printers, and telephone modems with little effort. Commodore's partial version of the IEEE-488 has its own name: the *serial bus*. Three other communications paths are supported by the Commodore 64, but are permanently dedicated to just one peripheral device each. These paths service the data cassette, the keyboard, and the TV screen. We will discuss the characteristics of each of these five paths individually.

IEEE-488/Serial Bus. The IEEE-488 standard defines a path that connects a computer with one or more peripheral devices through a data bus. The IEEE-488 path is often simply called the *IEEE bus*.

In the IEEE-488 bus, binary data travel over eight data bus wires simultaneously, one byte at a time. This is called parallel transmission. At a given transmission rate *parallel* transmission is eight times faster than sending data one bit at a time, or *serially*. However, serial paths require fewer wires and are less expensive.

The IEEE bus differs another way from most communications paths. A path usually allows only one device to be attached at each end. The IEEE bus can connect multiple devices along its length. Since a link has been defined as a single device at-

tached to a path, it is clear that the IEEE bus can support many links at once. This will lead to some interesting possibilities.

The IEEE-488 path was originally designed to attach test equipment such as voltmeters and oscilloscopes together for remote control and monitoring. This is still its most common use. However, Commodore has also made use of the bus to connect a series of business-quality peripherals to some of its other computers, although not to the Commodore 64.

Economy has dictated the Commodore 64's departure from the IEEE-488 standard. Most of the IEEE rules have been obeyed, but a less expensive bit-wide or serial data path has been substituted. Commodore calls the new path the *serial bus.*

IEEE-488 peripherals will not work with this path, so Commodore built special C64 peripherals that will. Among them are familiar devices such as the 1541 disk drive and the 1515 printer. Many but not all C64 peripherals attach to the computer through this bus. Communications with the disk drive are relatively slow because serial data transmission causes a bottleneck. Only about 300 bytes can be transmitted per second. However, the serial bus is easy to use because the kernel handles its functions for us.

Full IEEE-488 communications can still be performed by the Commodore 64 if an IEEE-488 interface card is attached to the computer. However, the complications in coordinating IEEE-488 bus activity to the C64's internal timing requirements makes using most of these cards a matter of very advanced and extensive programming.

A better alternative is to attach a *bus converter* to the computer. This type of device converts the standard signals on one type of communications path to standard signals of another. In this case, the conversion is from the control method and serial data of the serial bus to the control method and parallel data of the IEEE bus, and vice versa. Of course, the maximum speed for data transfer over the IEEE bus must be restricted to the maximum speed of the serial bus for the conversion to work. The programmer uses the same kernel calls and machine code that he or she used with the serial bus alone. One bus converter is the Interpod, made by Oxford Computer Systems of Oxford, England, and available in the United States.

A faster-operating variation on this type of bus converter plugs into the cassette port to gain access to the C64's internal buses and then substitutes for and makes itself appear like the serial bus to the kernel. It makes the conversion to and from the IEEE-bus communications without the speed limitations of physically going through the serial bus. An excellent example of this type of converter is the BusCard II by Batteries Included, which is available from retailers or from their office in Irvine, California. This unit is more expensive than the Interpod, which is the trade-off made for the unit's extra speed, a built-in machine language monitor, and a BASIC 4.0 interpreter. It is a well-built and easy-to-use device and a good standard of comparison for evaluating other bus converters. It also has a serial-to-Centronics bus converter, which allows using non-Commodore printers as if they were attached to the serial bus.

Commodore makes a line of peripherals that are faster and more robust than

those of their Commodore 64 line. These peripherals are designed to work with Commodore business computers via the IEEE bus. They include disk drives and printers, and can be used with the Commodore 64 through a serial-to-IEEE bus converter like the Buscard. The bus converter allows a 33% speed increase over the serial bus in transferring data to and from a disk drive. This could be important to someone who routinely writes and assembles large programs, for instance, because assembling a large program requires moving a lot of data back and forth between the computer and the disk drive. Using the Commodore 64's serial bus and a 1541 disk drive a long program can take as long as a half hour to assemble. The 1541 has an annoying tendency to overheat during such prolonged use, often causing irreparable damage, unless extra steps are taken to cool the drive (e.g., with a fan directed at the drive).

The Commodore business drives are better than the 1541 but they have their own disadvantages, namely expense and incompatibility with the copy protection schemes designed for the 1541 drive. By accessing special characteristics of the 1541 drive these schemes prevent many purchased programs from being copied onto backup disks and, incidentally, prevent the same programs from running on the Commodore's business drives.

A preferable 1541 replacement drive, one of lower cost than the Commodore business peripherals and that works well with most but not all copy protection schemes, is the Super Disk available from MSD Inc. of Dallas, Texas. The Super Disk can be used directly on the serial bus or in combination with the Buscard on the IEEE bus for faster data transfer rates.

Both Commodore and MSD produce a dual drive configuration of their 1541 replacement drives. For serious software development a dual drive 1541 replacement, particularly with a serial-to-IEEE-488 bus converter, makes for a durable and fast system that simplifies the frequent disk backups necessary to protect your work. This is because in the dual configuration both drives have the same device number, 8, with the two drives being numbered 0 and 1 in ASCII messages sent to the drives. Device numbers and drive messages are discussed in detail later in this chapter. Sharing the same device number allows the data on a disk in drive 0 to be copied directly onto a disk in drive 1 with a single BASIC DUPLICATE or BACKUP command. This is much quicker and easier than a backup operation with 1541 drives, which requires loading files into memory from the original disk and saving them to the backup disk one at a time.

Describing the complexities of actual IEEE-488 bus activity is beyond the scope of this book, so if you are interested in using the IEEE bus the hard way (i.e., without a bus converter) or just in learning more about it, I suggest you obtain a copy of one of the IEEE bus tutorials on the market and enjoy.

RS-232. The second communications path, RS-232, also connects a computer to a peripheral device through a data bus. However, it is more typical in that only one peripheral can be attached to the far end of the bus at a time. Binary data are transmitted serially.

The Commodore 64 provides both the physical signals and kernel support for an RS-232 path. However, the voltage of the C64's signals is too low for RS-232 peripherals, so you must at the least purchase an RS-232 board that plugs into the C64 to use these devices.

Most microcomputer peripherals on the market communicate over an RS-232 path. Since Commodore's peripherals use a serial or IEEE-488 path, you will probably never need to use an RS-232 peripheral. If you should desire to use one, however, you have the same two options as with IEEE-488. First, you can obtain a bus converter to bridge the serial and RS-232 buses. Interpod contains one of these converters. As with a serial-to-IEEE bus converter, an RS-232 bus converter should let your programs communicate with RS-232 devices using the same kernel calls and assembly language code as they use with devices on the serial bus. Be sure to check any bus converter's user's manual before purchase to make sure that it works this way.

The second option is to use a less-expensive board that converts the available RS-232 signals to the correct voltage. This is a more difficult alternative since programming for these boards forces you to directly manipulate the RS-232 locations in the memory map. Most people should avoid this alternative. However, if you enjoy a challenge or simply appreciate pain, your reference for direct RS-232 programming will be the Commodore's *Programmer's Reference Guide.* It gives the minimum and only available information on these locations in its RS-232 and detailed memory map sections.

Data Cassette. The C64's third type of communications path is dedicated to the data cassette. This path carries data serially, like the serial bus, but is otherwise simpler since it services only one peripheral. Its speed is limited by the data cassette, which as any data-cassette user knows is almost interminably slow. Any other physical characteristics are unimportant to the programmer because, unlike the IEEE-488 and RS-232 paths, the cassette path is completely supported by the kernel and C64 circuitry.

Keyboard and Screen. The last two paths are also physically dedicated to just one peripheral device each. The keyboard path carries data from the keyboard to the computer. The screen path carries data from the computer to the TV screen. Recall that these paths are contained in the kernel's default channels, the channels that are selected whenever a program has not substituted channels of its own choosing.

The screen path normally carries data in a 40-character-per-screen-line or 40-column format. A typical typed page can have up to 80 characters per line, but a normal TV set cannot show that much detail. So word processing programs for the C64 wrap around each line in a document to show its two halves on two lines. However, something like a bus converter is available that converts this format to an 80-column format that shows an entire page across. These devices, called *80-column cards,* are used with computer monitors that can clearly display a full 80 columns.

The B.I.-80 card from Batteries Included of Irvine, California, is a high-quality example of this type of converter.

Since these paths are fully supported by the kernel and C64 circuitry, no further knowledge of their physical characteristics is needed to use them.

The first three C64 communications paths all connect an automated machine, the computer, with an automated machine that is a peripheral. Data are moved between the machines as simple files, leaving the interpretation of the transferred data to the machines (i.e., their internal programs or circuitry). Thus the serial/IEEE-488, RS-232, and cassette buses are called the *file paths.*

The last two paths connect an automated machine, the computer, with a machine that is directly controlled and used by a human being. The file format is not well suited for human input or viewing, so the kernel intervenes and places input and output data into sentence and character formats that are more natural to human beings. The give-and-take interaction between user and computer that these paths support leads them to be called the *interactive paths.*

The kernel handles channels containing file paths differently than channels containing interactive paths. When we discuss how channels are used in programs, we will cover the types of channels in each of these categories separately.

The tree diagram in Fig. 5.6 summarizes the communications paths available to the Commodore 64.

Peripheral devices. Peripherals expand your computer into the world around it. They allow you to automate a broad range of interesting tasks, from running your own weather station to controlling home lights and appliances by timer, electric eye, or voice command.

You cannot use peripherals of which you are unaware, so a reasonably complete list of what is available follows. It includes devices that attach to any of the C64's five communications paths, to the paths provided by bus converters, and to the C64's internal buses through the cartridge port. This list consists of floppy and hard (high-capacity) disk drives, cassette drives, dot matrix and letter-quality printers, telephone modems, pen plotters, televisions, high-resolution video monitors, high-fidelity audio equipment, serial-to-IEEE-488 bus converters, serial-to-RS-232 bus converters, serial-to-Centronics parallel (printer) bus converters, RS-

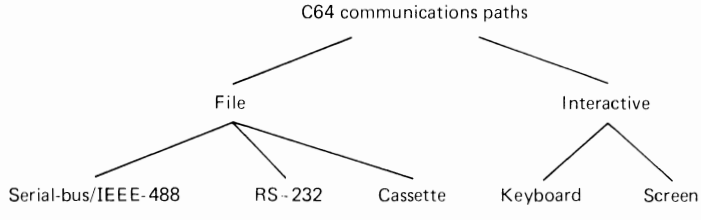


Figure 5.6

232-to-Centronics bus converters, lab measurement instruments such as oscilloscopes and pressure sensors (this is too large a category to list in its entirety here), weather gauges, voice recognizers and synthesizers, burglar and fire alarms, remote controllers for household devices, PROM (user-programmable ROM) programmers, radio transmitter/receivers, optical transmitter/receivers, television transmitter/receivers, television cameras with binary output, robots, A/D and D/A converters (recall the discussion in Chapter 1) and the myriad electronic devices that can communicate with them, keyboards with add-on music synthesizers, game boards with piece-position sensors (chess, checkers), bar code readers, character/image readers, teletypes, paper-tape punches and readers (often part of teletypes), and bus expansion boxes for attaching many peripherals to the computer at once.

From the programmer's viewpoint, each of these peripherals is accessed through one of the five C64 communications paths. The data-cassette, keyboard, and TV are inseparable from their communications paths and can be used with only the minimal programming required to run those paths. Serial-bus and RS-232 devices demand more individual attention; they are more complex, and in many cases must be commanded into specific states before, during, and after their use.

The standard for a communications path defines a general method for communicating with any device attached to it. Thus, rather than try to discuss how each of the thousands of available peripherals is used, we will look at each of the communications paths to determine the general programming requirements of their peripherals.

IEEE-488/Serial Bus. IEEE-488 bus devices can be discussed as a group that includes the serial bus devices, since both are similar from the programmer's point of view. Each IEEE-488 device has its own bus address, which must be sent out the bus to select the device for use. This address may be preset by the manufacturer, or it may be adjustable by the user. Bus addresses are totally unrelated to memory map addresses.

An IEEE bus address has two parts, one mandatory and one optional. The mandatory part of the address is the *primary address,* which is a number from 0 to 31. This address is usually sufficient to represent and select a device. It is sometimes called a *device number* since it identifies and selects one device on the bus.

The optional part of the address, a *secondary address,* also from 0 to 31, can be combined with the primary address to provide up to 961 addresses (31 × 31). Secondary addresses are not usually needed to select devices, since the IEEE-488 standard allows no more than 15 devices to be attached to the bus at a time. The C64's serial bus limits it even further, to five devices at a time. Interpod allows a combination of up to 30 devices on the IEEE-488 and serial buses. All these limits are less than the 31 devices allowed by a primary address, so the secondary address is freed for a different use: carrying commands to the primary-addressed device. The secondary address is sometimes called a *command byte* because of this role.

For example, the primary address of a single Commodore 1541 disk drive on

the serial bus is normally "8." A secondary address value between 2 and 14 commands the disk drive to prepare to transfer data with the computer. So, assuming that the disk drive is attached to the serial bus, if a program calls the appropriate kernel modules to send a primary address of "8" and a secondary address of "8" out the bus, the drive will activate itself and prepare to transfer data with the computer.

A 1541 disk can store up to 170K bytes of data in from 1 to 144 files, or independent data structures.

RS-232. Only one device can be attached to the RS-232 path at a time, so devices need no bus address. However, the RS-232 standard allows several options on how data are physically handled, which can vary from one peripheral to the next. These options must be accounted for by a program using the RS-232 bus. Generally, the program adjusts the computer or bus converter to match the characteristics of the peripheral.

In the RS-232 standard, data are sent serially at a selectable transmission rate, with the data divided into groups of seven or eight bits followed by a selectable number of *stop bits* signifying the end of the group, and with one of several types of parity assigned to the groupings. Each data group may also be preceded and/or followed by "no data" 1 values that tell the receiving device when valid data are on the path.

The first quantity, transmission rate, governs the length of time each bit exists on the RS-232 bus. Common transmission rates are 110, 300, 1200, 2400, and 9600 bits per second, or *baud.* The devices on both sides of the bus must be "told" the same transmission rate so that the receiver can read the value on the data line at the same rate that the transmitter is writing the data.

The last three quantities relate to the low-level physical structure of RS-232 data. A single data-bit group is isolated in Fig. 5.7 by surrounding "no-data" 1 values.

The RS-232 data line is normally at the binary 1 value. When a data unit is transmitted, the line changes to a 0 for one bit length of time at the current transmission rate. The data receiver uses this start bit to synchronize itself with the data transmitter.

Seven or eight bits of data then follow on the line. One of those bits can be reserved for parity checking, which we described in Chapter 1. Parity can be "even," for the parity bit being set to 1 as needed to maintain an even total number of 1's in the data word; "odd," to maintain an odd number of 1's in the data word; or "none," for no parity checking.

Finally, one, one and one-half, or two stop bits of binary value 1 are transmitted. "One and one-half" stop bits simply means that a 1 value is held on the line for

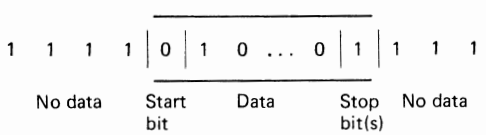


Figure 5.7

ferent communications needs. However, the data characteristics of the devices at the start bit of another data unit begins immediately after the stop bits.

The RS-232 standard allows for varying these quantities to account for different communications needs. However, the data characteristics of the devices at each end of a given path must exactly match.

As we have said, matching is normally done at the computer end of an RS-232 link. The RS-232 characteristics of the C64 or the bus converter, the only two devices that can be attached to the closest end of the RS-232 path, can both be altered by a program. So we can program the computer or bus converter to communicate over the RS-232 path with even, odd, or no parity, 7- or 8-bit data, and so on. This allows a program to match the computer end to the peripheral end of an RS-232 path and thus complete a working RS-232 link.

The manual for any RS-232 peripheral device should include a description of its data and timing characteristics. Similarly, the C64 or RS-232 bus converter manual should include instructions for changing its data and timing characteristics. Changing the converter to match the device should be simple to understand and to do. For example, the manual for the Interpod shows that the converter's RS-232 characteristics can be altered by sending a few bytes out the serial bus. This is a basic channel-I/O operation that we will learn to do shortly. Any other good RS-232 bus converter must also rely on basic I/O operations for setting its RS-232 characteristics.

Without a bus converter, the RS-232 characteristics must be changed in the computer. This is done by altering the RS-232 memory map locations. The information necessary to do this is found in the RS-232 section of Commodore's *Programmer's Reference Guide.*

Data Cassette. The only peripheral that ever attaches to the cassette path is the data cassette. Because the data cassette is used for data and program storage in much the same way as a disk drive, the kernel is designed to make using the data cassette as similar to using a disk drive as possible. The data cassette is accessed totally through the kernel, so unlike IEEE and RS-232 devices, it has no physical characteristics that must be handled separately.

Keyboard and Screen. The keyboard and screen devices are also completely handled by the kernel. You have a slight choice in the type of screen device that can be attached, however. A standard TV set is usually used with the C64, but sharper and clearer video monitors that work with the screen path just like a TV set are also available.

Links. Communicating with the outside world requires a combination of a communications path and a peripheral. As we said earlier, such a combination is called a *link.*

The kernel defines its own links for the default keyboard and screen channels. Any links created by a program must be defined for the kernel by that program. The

kernel keeps track of these user-defined links by placing the link definitions in a table of up to 10 entries.

The link-definition area is called the *open table* and is located starting at address 0259h. Each link definition consists of three bytes which are provided by the executing program. The first byte in each three-byte group is the *link ID number*. A program refers to a link definition by this number when telling the kernel to activate the link to form a channel or to dissolve the link by removing it from the open table. Commodore calls this first byte the *logical file number* or lfn, for reasons that will become clear later. The second byte is a device number which identifies the peripheral device and communications path being used. The third byte is a command to the device or the kernel for when the link is activated into a channel. If no command need be sent, this byte can be given a dummy value.

Conceptually, the open table is of the form shown in Fig. 5.8. Links are added to the table in the order that they are defined, from position 1 through position 10.

The open table is physically stored as 10 consecutive lfn bytes starting at location 0259h, followed by 10 consecutive device-number bytes starting at 0263h, followed by 10 consecutive command bytes starting at location 026Dh. So if the kernel needs to read the link definition whose lfn is 8, and if that definition is third in the open table, the kernel will read the third location in the lfn area, at address 025Bh, for the lfn, the third location in the device number area, at address 0265h, for the device number, and the third location in the command area, at address 026Fh, for the command byte.

Unused definitions in the table are filled with 0 values by the kernel. The kernel will not accept more than 10 link definitions from a program until definitions have been removed from the table to make room.

Each of the three bytes in a link definition is discussed in detail next.

Logical File Number. An lfn can be any value between 01 to 7F hex. It would be ambiguous to give the same lfn value to two different links, so the kernel will not accept an attempt to do so (we shall see how this works shortly).

Device Number. The device number identifies both a device and the path to which it attaches. This is done by dividing the possible device-number values from 0 to FF into five ranges, one for each communications path.

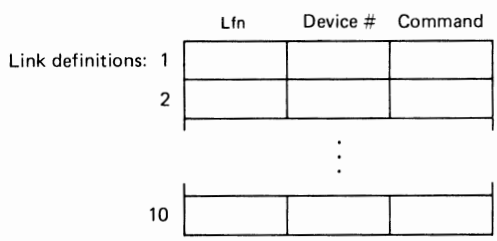


Figure 5.8 Open table

TABLE 5.6 DEVICE NUMBERS

Range	Path	Value	Device
0	Keyboard	0	Keyboard
1	Cassette	1	Cassette
2	RS-232	2	RS-232
3	Screen	3	TV/monitor
4–255	IEEE/serial	4–5	Printers
		8–11	Disk drives

The RS-232, cassette, keyboard, and screen paths each allow for only one attached device at a time, so only one device number is needed to select both path and device. Therefore, the number ranges for these paths contain just one value each.

The IEEE/serial-bus path can have many attached devices at once, so it has been allotted the remainder of the possible byte values. Device numbers in the IEEE/serial-bus range are used as primary addresses to select particular peripherals on the bus.

The device-number ranges for the five communications paths, and the specific device numbers that have already been assigned to particular peripherals, are shown in Table 5.6.

Commands. The command byte serves as a command to the device on the link or to the kernel on how to use the link. If the link contains a IEEE/serial-bus peripheral, the command byte will be sent to it as a secondary address. Where no command is necessary, a dummy value of FFh should be placed in the command byte.

Table 5.7 shows useful values of the command byte for the most common types of links. Note that the final meaning of many of these command values depends on how the link is used in channel I/O. Thus the "meaning" column of this table will remain meaningless to you until we have discussed the channel I/O operations.

Channel Programming

Using channel I/O distances the programmer from the physical considerations in moving data between the computer and a peripheral. The physical aspect of communicating with a peripheral includes details such as monitoring and controlling the voltage on individual lines in the physical bus, comparing physical changes on the bus with a clock to control the timing of other changes within predefined limits, and converting data in memory into a form compatible with the path, and vice versa. The kernel handles these physical details and provides other services so that we can treat communicating with a peripheral as a simple matter of setting up a channel, moving data through it, and closing out the channel.

TABLE 5.7 USEFUL VALUES OF THE COMMAND BYTE

Device	Value	Meaning
Keyboard	Not used	
Cassette	0	With whole-file I/O: If channel will be used for LOAD: 'Place file at address to be passed to LOAD in X and Y' If channel will be used for SAVE: 'Write file with EOF marker' With partial-file I/O: 'Create an input channel'
	1	With whole-file I/O: If channel will be used for LOAD: 'Place file at address it was SAVEd from' If channel will be used for SAVE: 'Write file with EOF marker' With partial-file I/O: 'Create an output channel'
	2	Only used for whole-file I/O: If channel will be used for LOAD: 'Place file at address it was SAVEd from' If channel will be used for SAVE: 'Write file with EOF and EOT'
RS-232	Not used	
Screen	Not used	
Serial/IEEE		
Printer	0–10	(Effect varies between models)
Disk drive	0–1	Used only with whole-file I/O: value effects are same as with cassettes
	2–14	'Create a partial-file data channel'
	15	'Create a command/error channel'

Recall that the five steps in performing channel I/O are

1. Create a link.
2. Activate the link.
3. Communicate via the channel.
4. Deactivate the channel.
5. Dissolve the link.

It is useful to group these steps into two activities: channel management and channel communications. Channel management consists of steps 1, 2, 4, and 5, which involve setting up and closing out channels. Channel communications consists of step 3.

Channel management and channel communications are performed in different ways depending on whether data will be moved through the channel using whole-file, partial-file, or interactive I/O. We will discuss channel management and channel communications in terms of these data movement methods.

Channel Management

The kernel provides one group of modules for channel management, and another group for channel communications. The channel management modules are introduced below. They will be used for managing the various types of channels we will discuss afterward.

Kernel support. The kernel modules used to create and terminate channels are SETLFS, SETNAM, OPEN, CHKIN, CHKOUT, CLOSE, and CLRCHN. As they execute, these modules can also provide feedback to the person running the program by displaying error and status messages on the screen. Error messages inform the user of any condition causing a kernel module to be unable to perform its task. This is particularly useful in debugging the program. Status messages inform the user of the kernel module's status in performing its task. Either, neither, or both types of messages are selected for all modules by executing a kernel module called SETMSG.

The channel management modules are summarized below. The "prerequisites" category refers to modules that must be called before calling the current module. The "altered registers" category lists the microprocessor registers that are altered during the module's execution. These registers must be saved before the module call and reloaded after the return if their data are to be preserved. Moving them through the accumulator into and out of the stack is usually the best method. Note that register Y is unaffected by most of the routines. This makes Y a good place to store a loop counter or other frequently used variable. The JSR address is the address of the module's JMP table entry; the program should execute a JSR to the JSR address to call the kernel module.

The other summary categories are self-explanatory.

SETLFS:
Purpose: Define a communications link.
JSR address: FFBA hex.
Prerequisites: None.
Data passed: Lfn in A, device # in X, command in Y.
Data returned: None.
Altered registers: None.

SETNAM:
Purpose: Name a file-path link.
JSR address: FFBD hex.

Prerequisites: SETLFS.

Data passed: Name of 1 to 16 XASCII characters, beginning at 'name_address' in memory, with low byte of 'name_address' in X and high byte of 'name_address' in Y, and byte length of name in A.

Data returned: None.

Altered registers: None.

OPEN:

Purpose: Add link definition to open table, prepare the peripheral to send or receive file.

JSR address: FFC0 hex.

Prerequisites: SETLFS (SETNAM if link is to a named file).

Data passed: None.

Data returned: Error #s 1,2,4,5,6,7,F0 in A (see SETMSG).

Altered registers: A, X, and Y.

CHKIN:

Purpose: Select a link to be input channel.

JSR address: FFC6 hex.

Prerequisites: OPEN.

Data passed: Lfn in X.

Data returned: Error #s 0,3,5,6 in A (see SETMSG).

Altered registers: A and X.

CHKOUT:

Purpose: Select a link to be output channel.

JSR address: FFC9 hex.

Prerequisites: (OPEN for any channel except keyboard or screen).

Data passed: Lfn in X.

Data returned: Error #s 0,3,5,7 in A (see SETMSG).

Altered registers: A and X.

CLOSE:

Purpose: Remove link from open table, close file in storage.

JSR address: FFC3 hex.

Prerequisites: Makes sense only after OPEN.

Data passed: Lfn in A.

Data returned: Error #s 0,F0 (see SETMSG).

Altered registers: A, X, and Y.

CLRCHN:

Purpose: Deactivate serial bus devices, terminate channels, set channels to defaults (keyboard and screen).

JSR address: FFCC hex.

Prerequisites: CLRCHN is only needed after channels have been created.

Data passed: None.

Data returned: None.

Altered registers: A and X.

SETMSG:

Purpose: Select type of message feedback from kernel modules.

JSR address: FF90 hex.

Prerequisites: None.

Data passed: Desired message type, in A:

0 = no messages
4 = error messages (e.g., I/O ERROR #5)
8 = status messages (e.g., SEARCHING FOR filename)
C = error and status messages.

Data returned: None.

Altered register: A.

SETMSG note. Kernel error messages are displayed in the form of the words I/O ERROR # followed by a number. If error messages have not been selected, the error number is simply returned by the kernel in the accumulator. The meaning of each number is shown in Table 5.8, followed by the names of the kernel modules most likely to draw each type of error (including channel communications modules).

Kernel status messages are short phrases that inform the program's user of what to do next and what the kernel is doing. An example of the former type of mes-

TABLE 5.8 KERNEL ERROR MESSAGES

Error	Meaning (modules likely to draw error)
0	STOP key halt (LOAD, CHKIN, CHKOUT, CHRIN, CHROUT, CLOSE)
1	Open table is full (OPEN)
2	Link already exists (OPEN)
3	Link not in open table (CHKIN, CHKOUT)
4	File not found on storage device (LOAD)
5	Device not on path or EOT (OPEN, LOAD, SAVE, CHKIN, CHKOUT)
6	Input device expected (OPEN, CHKIN)
7	Output device expected (OPEN, CHKOUT)
8	Missing file name (LOAD, SAVE)
9	Illegal device number (LOAD, SAVE)
F0 (240)	RS-232 buffer error (OPEN)

sage is 'PRESS PLAY ON CASSETTE'. An example of the latter is the 'SEARCHING FOR file name' message that appears when the kernel is searching for a file on disk or cassette. Since the control messages are user oriented and self-explanatory, we need not list them here.

Whole-file channels. In Chapter 2 we said that one meaning of the term *file* is "repetition data structure." The term "file" has a looser definition in operating system terminology; it means any data structure transferred to or from the computer.

The simplest type of channel to manage is one that will be used to transfer an entire file to or from a peripheral at once. A file is transferred as an entirety using *whole-file I/O*. Channel management is simplest with whole-file I/O because once the file source and file destination have been defined and the transfer has been commanded, the kernel performs the transfer and even closes out the channel without further program action. Thus the only channel management needed with whole-file channels is to define a link. This requires calling at most only two kernel modules: SETLFS and SETNAM.

Whole-file I/O can be used with only two peripherals: the data cassette and the disk drive. Managing whole-file channels to either of these devices requires the same assembly language instructions and kernel calls. The necessary program code is shown below. However, different parameters are passed for cassette I/O than for disk I/O, and there are several other important differences between the two, so the specifics for each device will be discussed afterward.

```
;prepare a link for whole-file I/O with cassette
    ;define a link
        LDA #lfn
        LDX #device-number
        LDY #command
        JSR SETLFS

    ;identify the file (optional—can be omitted)
        LDA #number of bytes in name
        LDX #low-byte of address of first byte in file name
        LDY #high-byte of address of first byte in file name
        JSR SETNAM
```

The assembly code above can be preceded by a call to SETMSG to select the type of user feedback that the called modules will provide.

Preparing to store or retrieve a file on a storage device first requires defining a link to that device using SETLFS. The name of the file can then be specified by passing the file name to the module SETNAM from somewhere in RAM. This is usually done by including the name in the program as reserved bytes—for instance, using a BYTE or similar program directive, as discussed in Chapter 3.

No further channel management operations are required for whole-file I/O. Setting up and executing the appropriate channel communications module automatically terminates the channel when the transfer has been completed.

Data Cassette. One specific of whole-file channel management with the data cassette is the choice of parameters passed to SETLFS. The lfn can be any value from 1 to 7F hex. The device number is always 1. The command number can be from 0 to 2 to select file-handling options that we can now explain.

The command byte is used to prepare the kernel for the upcoming file transfer. The file-handling option selected by the command byte is exercised during the transfer itself. Therefore, the meaning of the command byte depends on the type of transfer that will be performed. An output transfer is called a *save* and an input transfer is called a *load.*

If the file is to be saved on the cassette, a command byte of 1 will cause an EOF or end-of-file marker to be written at the end of the file when it is transferred. The kernel will use this marker when loading the file to determine when the entire file has been transferred. A command number of 2 followed by a file save causes both an EOF marker and an EOT or end-of-tape marker to be written after the file. The EOT marker effectively ends the tape. When the kernel is searching for a file during a whole- or partial-file input operation and it encounters the EOT marker, it will stop its search and return control to the calling program.

If the file is to be loaded from the cassette, 0 and 1 command values select where the kernel will obtain the beginning address of the memory area in which to place the file. A command value of 0 tells the kernel to look for the beginning address in the X and Y registers when the kernel module LOAD, which loads the file, is called. A command value of 1 tells the kernel to place the file into the same locations from which it was originally saved. The details of whole-file channel communications (i.e., loading and saving files) will be covered in the channel communications section.

With the cassette, identifying the file name by setting up and calling SETNAM is optional. A file can be saved without a name, and performing a load operation without specifying a file name results in the kernel obtaining the first file it comes to on the tape.

A file in memory can be saved with the same name as a file already existing on the cassette. When the data are saved, a new file of the same name is created starting at the position at which the tape is currently located, overwriting whatever was on the tape previously. The kernel does not check the tape for another file of the same name. This situation is treated differently when the file is being stored on disk, as the next section explains.

Disk Drive. The parameters passed to SETLFS for disk files are as follows. As always, the lfn can be from 1 to 7F hex. The device number depends on which one of how many disk drives are being accessed. Recall that Commodore 1541 disk drives are delivered with a device number of 8. Most C64s use only one drive. Nevertheless, up to four drives can be attached, and the additional drives are given device numbers 9 through 11. Command numbers of 0 and 1 have the same effect as with the data cassette and should be used accordingly.

The name of disk files must be specified so SETNAM must be called. This requires an expanded name format. To access a file on drives 9 through 11, a device number between 9 and 11 must be passed to SETLFS, and the XASCII file name must be preceded by a drive number from 1 to 3 in XASCII code, respectively, and a colon. To compensate for the extra characters, the 'file name length' value passed to SETNAM must be increased by two, and the 'file name address' values must point to the drive-number byte instead of the first byte in the file name itself.

Thus, depending on which drive is being accessed, the file name passed to SETNAM will be in one of the following two formats:

<dr>:<filename> OR
<filename>

For instance,

1:test

identifies the file 'test' on the drive at device number 9, and

test

identifies the file 'test' on the drive at device number 8.

A channel should not be created for output to a file that already exists on disk. When the output data transfer is initiated (discussed in the channel communications section), the kernel will check for the existence of the file on disk and then abort the transfer. The attempt will place the disk drive into an error condition that the user or program must take additional steps to correct (see the discussion of the disk command/error channel in the partial-file I/O portion of the channel communications section).

Each disk contains a directory or list of the files it contains. To read it into memory with whole-file I/O, a link is defined by passing SETLFS the device number of the drive and the command value 0 or 1, and by passing SETNAM the name

$<dr>

The drive number dr ranges from 0 to 3 (ASCII code) for drive device numbers 8 through 11, respectively. Passing SETLFS a 0 command tells the kernel to place the directory at an address passed to the LOAD module in registers X and Y. Passing SETLFS a 1 value causes the directory to be loaded at address 800 hex. This is fully explained in the whole-file I/O portion of the channel communications section.

The drive number part of the directory name can be omitted for drive 0 (device number 8), so that the name becomes simply $. Thus, passing SETLFS the device number 8, the command value 1, and passing SETNAM the file name $ prepares a link for reading the directory of the first C64 drive into memory starting at address 0800 hex. A device number of 9, a command byte of 1, and a file name of $1 will prepare for reading the directory of the second drive into location 800 hex, and so on.

The directory is in the form of a series of 32-byte-long entries, each containing a file name followed by a file type. The file name is the 1- to 16-byte XASCII name passed to SETNAM, without the optional drive number and colon. The file type indicates to the operating system whether the file was created using a whole-file or one of two varieties of partial-file channel. This notifies the operating system of the channel-communications operations that can legally be performed on the file.

Files created using a whole-file channel are given type PRG, which reflects their common use for holding programs and can be accessed with whole-file I/O or a type of partial-file I/O called *sequential I/O,* which is introduced in the next section.

Partial-file channels. Partial-file channels are used to transfer a portion of a file at a time. Managing a partial-file channel requires more program action than whole-file I/O. Whole-file I/O was unusual in that we did not even have to complete the first step in channel I/O: creating a link. However, to prepare a partial-file channel we will have to complete channel I/O steps 1 and 2, where step 2 is activating the link. Further, since the kernel does not know when the transfer will be complete, it cannot terminate the channel automatically; the program has to terminate the channel using additional channel-management modules.

Partial-file channels are created using kernel modules SETLFS, SETNAM, OPEN, and CHKIN or CHKOUT. Partial-file channels are terminated using modules CLRCHN and CLOSE.

There are two varieties of partial-file I/O: sequential-file and relative-file. The characteristics of these I/O types are discussed in the channel communications section. The devices with which sequential-file I/O can be used include the data cassette, the disk drive, IEEE-488/serial bus devices, RS-232 devices, the keyboard, and the TV screen. Relative-file I/O can be used only with the disk drive. Both types of partial-file channels are managed using the same assembly language instructions. This assembly code is shown below. The parameter-passing and other differences between managing sequential-file and relative-file channels are discussed in the following disk-specific section.

A sequential-file link is defined with SETLFS and SETNAM, as was the whole-file link. The kernel module OPEN is called next, which causes the link to be placed in the open table and, if the link contains the disk or cassette, housekeeping duties to be performed on the device. These actions complete the first step of channel I/O, which is 'create a link'.

Next, one of two modules is called to activate the link as a channel. The kernel module CHKIN is called to create an input channel, while the module CHKOUT is called to create an output channel. This completes the second step of channel I/O, activating a link into a channel. The channel is then ready to carry data to or from the cassette.

The complete assembly code for creating a sequential-file channel is shown below. As before, it may be preceded by a call to SETMSG.

```
;create a sequential-file channel
  ;create a link
    ;define the link
        LDA #lfn
        LDX #device-number
        LDY #command
        JSR SETLFS          ;address FFBA hex

    ;identify the file
        LDA #number of name bytes
        LDX #low-byte of file name address
        LDY #high-byte of file name address
        JSR SETNAM          ;address FFBD hex

    ;add link to open table and do file housekeeping
        JSR OPEN            ;address FFC0 hex

  ;activate the link to form a channel
        LDX #lfn
        JSR CHKIN           ;input channel-address FFC6, OR
            CHKOUT          ;output channel-address FFC9
```

Channel-communications operations can then begin. Once the program has completed all such operations, the channel is terminated using the following assembly code:

```
;terminate the sequential-file channel
    ;deactivate the channel
        JSR CLRCHN
    ;dissolve the link
        LDA #lfn
        JSR CLOSE
```

Executing CLOSE removes the link from the open table and if the file is on disk or cassette, performs any file housekeeping necessary to make the file available for future use.

Sometimes programs have a reason to alternate between two or more input channels or between two or more output channels. You could execute all the channel

management steps above each time a particular channel is used, but that is inefficient.

Instead, a number of links can be created and kept in the open table by executing all the channel-creation instructions up to but not including CHKIN or CHKOUT for each link. A program selects the input channel and the output channel at any given time from the links in the open table, by setting up and executing CHKIN or CHKOUT. When finished with a particular channel, the program can deactivate it without removing it from the open table by executing CLRCHN but not CLOSE. After the last time the channel is used, however, it *must* be completely dissolved by executing CLOSE. This signals the kernel to perform housekeeping work that will make the file accessible to the kernel in the future. If you omit this step with a cassette or disk channel, you can lose the contents of your file.

Data Cassette. The parameters passed to SETLFS for a cassette link are as follows. The lfn can be between 1 and 7F hex. The device number must be 1. A command byte of 0 signals the kernel that an input channel will be created, while a 1 or a 2 signals the creation of an output channel. A command byte of 1 will also cause the file written through the channel to be written with a trailing EOF marker, and a 2 will cause the file to be written with both an EOF and an EOT (recall the command chart in the link section and the discussion of command numbers in the cassette portion of the whole-file section).

When the kernel module OPEN is called, a command byte of 0 causes the kernel to search for the file on the cassette, while a command byte of 1 or 2 causes the kernel to create a file on cassette. As with whole-file channels, cassette files need not be named and SETNAM is optional. If SETNAM is not used, a created file will be nameless. Its position on the tape should be written down so that it can be read later by rewinding the tape to a position just before the file and creating a nameless input channel to the cassette.

When the link is activated into a channel using CHKIN or CHKOUT, the channel direction chosen must be consistent with the direction chosen by the command byte.

Disk Drive. Both the sequential-file and relative-file methods for partial-file I/O can be used with the disk drive. We discuss channel management for these two I/O methods separately below.

Sequential Files. The link parameters passed to SETLFS are as follows. The lfn is any number between 1 and 7F hex. The device number is between 8 and 11, depending on which drive is being accessed. The command byte is any number between 2 and 14.

The file name format passed to SETNAM is in a slightly different format than with previous channel types. The format used with the disk drive is as follows:

<dr:><file name>,<S>or<P>,<R>or<W>

As usual, the dr: field is optional with a disk of device number 8. The file name portion is composed of from 1 to 16 XASCII characters. The third field, consisting of ASCII ",S" or ",P", indicates the kind of file that will be accessed. ",S" is used to create a file through the channel or to read from an existing file that was created through a sequential-file channel earlier. Such files have type SEQ in the disk directory. ",P" is used to read a PRG (whole-file) file through a sequential-file channel using sequential-file I/O. There is seldom any reason to do this, however. The fourth field, consisting of ",R" or ",W", indicates the direction of data transfer. ",R" selects Read, which is for data input, and ",W" selects Write, for data output.

Disks usually contain more than one file. Up to two individual links can be created for each file—one for input and one for output—but no more than five links total can be open (i.e., in the open table) to a single 1541 disk drive at once.

A special type of sequential-file channel can be created to send commands to the disk drive and to read error information from it. This channel is called the *command/error channel.* Its uses are discussed in the channel communications section, but its management is our concern now. The command/error channel is managed using the standard sequential-file assembly code, except that the channel is unnamed and the SETNAM preparation and call is omitted. The link is created with any lfn from 1 to 7F hex, the device number of the disk drive (usually 8), and a command number of 15.

The command/error link for each disk drive must be created before any other links to the disk are created, and terminated after all other disk links have been terminated, or many I/O operations will not work properly.

Relative Files. Relative-file I/O is unique to the disk drive. Like sequential-file I/O, its uses are explained in the channel communications section. However, to explain how relative-file channels are managed, we must note here that a relative file consists of from 1 to 720 equally sized records each from 1 to 254 bytes in length.

The main difference between managing a relative-file channel and a sequential-file channel is in the file name format. One of two file name formats is used. If the channel is to a file that already exists on the disk, a simple file name of from 1 to 16 XASCII characters is passed to SETNAM. If the channel is being created to a file that does not yet exist on the disk, the following format is used:

<dr:><file name>,L,<record length>

where 'record length' is the number of bytes in each record, and is stored in binary, not XASCII, code. The other bytes are in XASCII code as usual. Some example file names are:

```
1:testfile,L,80  =  file on drive 9 with 128-byte records
oldfile          =  an existing relative file on drive 8
```

Files created using a relative-file channel have type REL in the disk directory.

A relative-file link can be activated as either an input or as an output channel using CHKIN or CHKOUT.

IEEE-488/Serial. Sequential-file channels are the only method we will use to communicate with IEEE-488/serial bus peripherals. File names are not used, so the instructions preparing data for and calling SETNAM are omitted from the channel-management code.

When the channel is created, the device number is sent out as a primary bus address, and the command byte is sent as a secondary address, as explained earlier in the discussion of the IEEE-488/serial bus as a communications path.

RS-232. The only special aspect of managing a sequential-file channel to an RS-232 device is that file names and the SETNAM module are not used.

Interactive I/O. The remaining devices, the keyboard and the TV screen, use an I/O method better suited than whole- or partial-file I/O to human interaction. The kernel automatically creates an interactive channel to the keyboard whenever the program-selected input channel is terminated. Similarly, the kernel automatically creates an interactive channel to the TV screen whenever the program-selected output channel is terminated. Alternatively, the program can create interactive-I/O links and channels using the sequential-file management code of the preceding section, with device numbers 0 and 3 for the keyboard and screen, respectively, and a command number of FF hex. Interactive channels are never named, so SETNAM is not used.

Summary. The channel management code for file-path (i.e., whole-file and partial-file) I/O is summarized in the table on pg. 245.

Channel management for the interactive paths is not shown since interactive channels are normally created by terminating all other channels, and otherwise are managed like sequential-file channels.

Channel Communications

The kernel modules used for communicating through channels are introduced below. They will be used in the following sections.

Kernel support. The kernel modules used to communicate through already existing channels are SAVE, LOAD, CHRIN, GETIN, CHROUT, READST, and indirectly, PLOT. These modules are summarized below.

CHANNEL MANAGEMENT ASSEMBLY CODE

```
;define a link
        LDA #lfn
        LDX #device-number
        LDY #command
        JSR SETLFS

;name the link
;(only used with disk or cassette data links)
;(be sure ASCII name is at name-address)
        LDA #name-length
        LDX #name-address low byte
        LDY #name-address high byte
        JSR SETNAM
```

```
;OPTION 1: WHOLE-FILE I/O

;create input channel,
;load file, and
;terminate channel with LOAD
    SET UP PARAMETERS
    CALL LOAD

          or

;create output channel,
;save file, and
;terminate channel with SAVE
    SET UP PARAMETERS
    CALL SAVE
```

```
;OPTION 2: PARTIAL-FILE I/O

;add link to open table
    JSR OPEN    ;and open file if disk
                ;or cassette link
;create a channel
    LDX #lfn                    ;appoint link
    JSR CHKIN or CHKOUT         ;as channel
                .
    (move data)
                .
;terminate input and output channels
;when done
    JSR CLRCHN  ;terminate channels
                ;and reset to defaults,
    LDA #lfn    ;and optionally remove
    JSR CLOSE   ;one or more links and
                ;close their files
```

SAVE:

Purpose: Create disk or cassette channel, save memory to 'PRG' file, terminate channel.

JSR address: FFD8 hex.

Prerequisites: SETLFS (SETNAM optional with cassette).

Data passed: Start address of memory block in page 0 with low address byte first, pointer to the page 0 locations in A, end address of memory block as low byte in X and high byte in Y.

Data returned: Error #s 5,8,9 in A.

Altered registers: A, X, and Y.

LOAD:
Purpose: Create disk or cassette channel, load PRG file to memory, terminate channel.
JSR address: FFD5 hex.
Prerequisites: SETLFS (SETNAM optional with cassette).
Data passed: 0 in A,
if secondary address defined in SETLFS = 0, then
low byte of destination address in X and
high byte of destination address in Y; else
if secondary address in SETLFS = 1, then
X and Y not used
(destination address = SAVE address).
Data returned: Error #s 0,4,5,8,9 in A (see SETMSG).
Altered registers: A, X, and Y.

CHRIN:
Purpose: Wait for and return a byte from the input channel.
JSR address: FFCF hex.
Prerequisites: Creation of an input channel.
Data passed: None.
Data returned: Input byte in A.
Altered registers: A and X.
Special comment: If input channel is keyboard, first call of CHRIN initiates editor. Editor waits for further keys until carriage return is typed, then CHRIN returns to program with first typed character in the accumulator. Subsequent calls return succeeding characters from the line, ending with the carriage return. Next call starts process over. Up to 88 characters can be typed before a carriage return. See further description in interactive I/O section.

GETIN:
Purpose: Get a byte from input channel or keyboard queue.
JSR address: FFE4 hex.
Prerequisites: Creation of an input channel.
Data passed: None.
Data returned: Input byte or no-data 0 value in A.
Altered registers: A, X, and Y.
Special comment: If input channel is keyboard, GETIN removes a byte from keyboard queue (10-byte FIFO structure at 277 hex). Keystrokes are loaded into queue by module SCNKEY during system interrupt, until buffer is full; then keystrokes ignored until a byte is removed. See full description in interactive I/O section.

CHROUT:
Purpose: Send a byte through the output channel.
JSR address: FFD2 hex.
Prerequisites: Creation of an output channel.
Data passed: Output byte in A.
Data returned: None.
Altered register: A.

READST:
Purpose: Determine status of I/O transfer.
JSR address: FFB7 hex.
Prerequisites: None.
Data passed: None.
Data returned: Status byte in A. Each bit position from d0 to d7 represents an I/O condition; bit value 1 means condition is present, 0 means not present. Byte equals 0 after a normal I/O transfer. Not all bits carry data useful for kernel-level programming. The bits and conditions that are useful to the kernel-level assembly programmer are:

Bit	Condition
d6	End of cassette or disk file, or end of relative file record
d7	Cassette end of tape, or serial/IEEE device not present

Altered register: A.

PLOT:
Purpose: Manage the cursor (see below under "data passed").
JSR address: FFF0 hex.
Prerequisites: None.
Data passed: Carry flag set to 1 to move cursor, to 0 to read its position. If carry set to 1, X coordinate in register Y and Y coordinate in register X.
Data returned: If called with carry = 0, X coordinate in register Y and Y coordinate in register X.
Altered registers: A, X, and Y.

Whole-file channels. Files of all types are used to hold pure data. Only PRG or program files are routinely used to hold programs as well. A program almost always needs to be loaded or saved as an entire unit. This, plus the simplicity of whole-file I/O, makes the whole-file channels a natural for program handling.

Program files are also ideal for data structures that are small enough to be loaded and saved whole in a reasonable length of time. Just what is reasonable depends on your patience level and on the storage device. A 20K disk data file is tolerable to most people, but 5K is closer to the limit with cassette.

The kernel creates a PRG file by transferring a continuous block of memory from the computer onto disk or cassette. The structure of the file is therefore the structure of the information in that memory block, although to move the file the kernel treats it as a repetition data structure consisting of multiple binary bytes. The kernel also adds preceding and trailing information that it uses in handling the file.

To move the data or program code in a block of memory into a PRG file on cassette or disk, a program calls the kernel module SAVE while passing it the beginning and ending addresses of the memory block. SAVE then stores the file together with the beginning address of the memory area from which it was saved.

Once a link has been defined with channel management modules SETLFS and possibly SETNAM, the following assembly code will cause the kernel to create a cassette or disk PRG file that mirrors the contents of a section of memory starting at 'begin_address' and ending at 'end_address'.

```
;'SAVE' ASSEMBLY CODE
;save from 'begin_address' through 'end_address' to PRG file

    LDA #low byte of begin_address    ;place value of
    STA page_zero_location            ;begin_address into
    LDA #high byte of begin_address   ;page zero
    STA page_zero_location + 1        ;

    LDA #page_zero_location           ;place pointer to
                                      ;begin_address in A
    LDX #low byte of end_address      ;place value of
    LDY #high byte of end_address     ;end_address in X and Y

    JSR SAVE                          ;create the PRG file
```

No follow-up assembly code is needed because SAVE does all necessary file housekeeping and channel termination after it finishes writing the file.

When a PRG file is loaded it is placed into a continuous block of memory whose starting address is specified in one of two ways. Defining the whole-file link with a command byte of 1 passed to SETLFS tells LOAD, the file-loading module, to use the address saved with the file. This causes the file to be loaded into the same memory area from which it was saved. Programs in PRG files are almost always loaded this way, since most programs can execute in only one place in the memory map.

Alternatively, defining the whole-file link with a command byte of 0 tells LOAD to obtain the starting address from the X and Y registers, with the low byte in X. The program places these values in the registers before calling LOAD. Relocatable programs or data structures that need to be loaded into different memory areas under different circumstances can be loaded this way. The program must also place a 0 in the accumulator before calling LOAD.

The code below assumes that a command byte of 0 was used to define the link. Thus the calling program will provide the beginning address. A command byte of 1 allows the middle two instructions to be omitted, since the beginning address of the memory area will be obtained from the file.

```
;'LOAD' ASSEMBLY CODE
;load PRG file to begin_address

    LDA #0          ;the mandatory 0 in the accumulator
    LDX #low byte of begin_address      ;X and Y are ignored
    LDY #high byte of begin_address     ;if secondary address = 1
    JSR LOAD
```

The address of the highest location loaded is returned in the X and Y registers, with the lower byte of the address in X. The address should be stored for later use if the file length is not already known.

Data Cassette. When LOAD reaches a cassette-tape EOT marker without finding the file name it is supposed to load, the module returns with a kernel error message number of 5 in the accumulator.

Disk. The LOAD and SAVE code above suffices for simple file handling. Some more-advanced techniques using whole-file I/O are discussed below.

The Auto-Start Loader. Like most computers, the Commodore 64 provides a way for its user to perform basic operating-system operations by typing in commands from the keyboard. In the C64, the BASIC interpreter translates the typed-in commands into kernel calls and their supporting instructions.

The user types a BASIC instruction called LOAD to load a PRG file into memory. If the loaded file contains a BASIC program, the user then types the BASIC instruction RUN to command BASIC to start interpreting the program. If the file contains a machine language program, the user types the BASIC instructions SYS < beginning address > to in effect JMP to the beginning of the program and start it executing. For the user's convenience, this second step in starting a machine-language program can be eliminated by first loading a PRG file containing a special program called an *auto-start loader,* and letting it do the work from then on.

To use the auto-start loader we must first understand how the BASIC LOAD instruction works.

The format for the LOAD instruction is

```
'LOAD file name,device #(,1)'
```

where 'file name' must be the name of a file of type PRG. If the ,1 option is chosen, the PRG file will load at the address the file was originally SAVEd at. Otherwise, it will be loaded at 0800 hex, the beginning of BASIC program memory.

Once the file is in memory, BASIC retrieves the *warm start vector,* an address

in locations 0302 and 0303 hex, and branches to the code to which it points. The warm start address is placed in these locations when the computer is powered up: It points to a routine which returns control to BASIC for the next user input.

If the program you have loaded begins in memory locations below 0302 and is continuous through 0302 and 0303 with its own beginning address in the latter two locations, the BASIC LOAD instruction will load the program and then route execution to the program's beginning. This is where the loader will reside.

The usual area for the loader is from 02A1 to 0303. Part of this area is dedicated to RS-232 communications, which are not used during a LOAD. The rest of it is free all the time.

The actions taken by a typical loader are:

1. Change memory map to assembly language.
2. Load the primary program.
3. Start the primary program executing.

The loader must originally have been SAVEd at the correct address (i.e., below $0302) and must be loaded to its SAVE address. This requires a secondary address of 1 in SETLFS. Assembly code for the basic auto-start loader is as follows:

```
;LOAD AND AUTO-START A PROGRAM

        * = 02A1
    ;change the memory map to assembly language
        LDA $01
        AND #$FE
        STA $01
    ;load the primary program
        LDA #$01 ;lfn
        LDX #$08 ;for disk drive, '#$04' for cassette
        LDY #$01 ;i.e., load primary program at its SAVE address
        JSR SETLFS
        LDA #primary-program-name-length
        LDX #name__address low byte
        LDY #name__address high byte
        JSR SETNAM
        ;LOAD the primary program with this section's LOAD code
    ;start the primary program executing
        JMP PROGRAM

name__address:      ;follows code because of memory-map req'ts.
        .BYTE 'program__name'
    ;loader address, placed over BASIC's warm start vector
        * = $0302       ;(locations between 'program__name' and $0302 are
                        ;meaningless filler)
        .BYTE $A1, $02  ;address of beginning of loader
```

So you can write a loader for each program on a disk, and start any primary program executing by typing in the following instruction from BASIC:

```
LOAD"loader_name",8,1
```

Overlays. If a program is too large to fit in the C64's memory, it can be divided into functional groups and placed into several PRG files. These files can be loaded as needed to execute the different functions of the program.

The basic function of the program (i.e., the boss module and possibly the central transform (see "Structured Analysis" in Chapter 4)) should be placed in the file that is loaded first. This file should remain in memory throughout program execution.

When the boss module reaches a function it cannot perform with any modules currently in the memory map, it loads the program file holding the necessary code into a separate memory area and continues execution. Similarly, when it reaches a function that requires data not in memory, it loads the file containing that data. Each time a new file is loaded it overwrites the old one in the memory area, which is why the name *overlays* is given to the transient files.

Overlays allow executing programs that would otherwise be beyond the capabilities of your computer. They can also be used with smaller programs to maximize the amount of free space in memory. This is sometimes done with word processing programs, for instance, to allow more room for the text entered by the user.

However, programs using this technique have the major drawback of executing much more slowly than programs that fit into memory whole. This problem can be eased by grouping the modules that will be used most often into the "heart" file, and by adjusting the size of the other modules for the best compromise between short load time and fewest loads required. These are conflicting goals since small files load in less time while larger files of logically grouped functions tend to remain in memory longer. Allowing for two or more independent memory areas for transient files helps, as does using as much of the memory map for the main and transient files as possible.

Other ways of getting around the problem of too-large files utilize sequential or relative files, which are discussed in the next two sections.

Partial-file channels. Partial-file I/O allows a program to transfer one byte, instead of an entire file, of data at a time to or from a peripheral. There are two types of partial-file I/O; sequential and relative, corresponding to the two file structures that can be used.

Sequential files can be transferred to and from all file-path devices, including cassette, IEEE-488/serial peripherals, and RS-232 peripherals. The kernel treats a sequential file as a basic data structure that can only be accessed sequentially starting from the first byte. The assembly code for sequential-file communications consists of a loop that reads or writes one byte at a time until the desired portion of the file has been moved. Because sequential-file I/O is the general case of partial-file I/O, we will examine its uses and code first.

Relative files can only be transferred to and from the disk drive. The kernel treats a relative file as a group of equally sized records. The records can be accessed

in any order, but the bytes within a record must still be accessed sequentially. This makes the data movement part of the assembly code for relative-file I/O similar to that used with sequential-file I/O. Additional code is needed to select a record before data are transferred. We will examine relative-file I/O in the upcoming disk section.

Sequential files are ideal for organizing data elements that can be processed in a sequential order and that are too numerous to fit in memory together or are numerous enough that the time delay in reading them all before starting execution is unacceptable. Storing such data in a sequential file allows them to be transferred only as necessary and allows the transfer time to be divided and distributed throughout the program's execution.

Since the direction of transfer is specified when a sequential-file channel is created, data can be transferred only in the first specified direction. Thus a program cannot read to a certain point in a file and then start writing data from there on, or any other such mixed read/write strategy.

Peripherals on the IEEE-488/serial and RS-232 buses send and receive data sequentially but generally do not store files. These devices use only sequential-file I/O, because it alone matches their communications needs. Their sent data are sometimes terminated with an end-of-transmission byte or bytes. A program can use this marker to tell when to stop reading the data. If no terminating pattern is used, the data can be thought of as forming a file of indefinite length, and a program must have its own criteria for ending the read.

Sequential files on cassette or disk are terminated by an end-of-file marker. A file-writing program can also divide a file into records and fields using reserved bytes to aid the file-reading process later. As we discussed in Chapter 2, these are the standard parts of a sequential data structure. This is particularly easy to do if the data are ASCII encoded, since so many byte values cannot occur as data (e.g., the value 00 hex). ASCII files created in BASIC use the carriage-return character (0D hex) to separate records, which it limits to 80 bytes length each, and the comma (2C hex) and colon (3A hex) characters to separate fields, which it also limits to 80 characters in length. Files can use any record or field separators that cannot occur as data; the only advantage in using BASIC's conventions is that BASIC programs will be able to read your sequential files properly. Record and field markers allow a program to read data record by record and field by field, although the file must still be read in sequential order.

A sequential file can be read or written straight through from beginning to end, but it can also be read in part and the read operation terminated. This allows the program to go back to the beginning of the file without finishing the entire file. For instance, suppose that you choose to keep the directory data structure of Chapter 4's phone directory task on disk but not in memory (although this is probably not a good decision). Suppose also that you order the listings by their frequency of usage, so that the most-accessed listing is first in the file, the second-most accessed is second, and so on. To find a given listing, the phone directory program starts at the beginning of the directory and reads listing records until it finds the

desired listing, which is generally before it reaches the end of the file. Rather than delay program execution by reading the rest of the file, the read operation can be terminated and the file reset by closing the file using channel management code. The file will then be ready for another search starting from its beginning byte.

Noncharacter (e.g., binary) data can also be stored in a sequential file. Such a file can have no records or fields, since any conceivable separator could also occur as part of the data.

A program reads partial-file data until a stopping condition is detected. That condition may be that the end of the file, a record, or a field is reached, or that a certain number of bytes have been read. The pseudocode for a partial-file read construct is as follows:

```
LOOP
        read a byte
    EXITIF stopping condition occurs
        store or write byte
ENDLOOP
```

The 'read a byte' step is executed by calling either of two kernel modules; CHRIN or GETIN. Both modules return a value in their accumulator. The main difference between the two modules in partial-file I/O is that CHRIN waits until a data byte is available and then returns it, while GETIN returns with either a data byte or a "no-data" value of zero immediately (there are other differences between the two in interactive I/O).

With a disk or cassette file, CHRIN's waiting is no problem. Data are available relatively quickly. With RS-232 or nondisk serial/IEEE devices, waiting can make a tremendous difference. CHRIN will hold up program execution, forever if necessary, until a data byte becomes available on the input channel. A good rule of thumb is to use CHRIN with disk or cassette drives, and GETIN with all other RS-232 and serial/IEEE devices.

The 'EXITIF stopping condition occurs' step is executed in different ways for different types of conditions. On reaching the end of a disk or cassette file, both CHRIN and GETIN continue to return a value as if there were still more data to read. However, a kernel module called READST can be called to detect the EOF condition. READST returns a 0 in the accumulator and zero flag after a normal, midfile data read. It returns a 1 in bit d6 of the accumulator if the end of the file has been reached with the previous data read. READST also checks for the EOT marker with cassette I/O, and sets bit d7 if it detects the EOT. A program can perform this EXITIF step with an EOF condition by calling READST and then executing a conditional branch. It can check for EOF or EOT by ANDing the returned byte with 80 or 40 hex, and then executing a conditional branch on zero.

The 'store or write byte' step requires either storing the value from the accumulator into an array in memory or writing it out another channel to a device such as the screen. As we will see, the latter requires executing the CHROUT module.

An end-of-record or end-of-field stopping condition can be tested for with a

CMP operation followed by a conditional branch on zero. A number of bytes condition can be tested for by setting up a loop counter, decrementing it each time a byte is read, and exiting the loop when the counter reaches zero.

We will show the assembly code for the EOF or end-of-file condition, and the 'store byte' option. We will assume that a disk file is being read, so that we need not test the READST return for EOT as well as EOF.

This assembly code is easily adapted for other stopping conditions, as explained earlier. To replace the 'store byte' step with a 'send byte' step, simply remove the four 'store byte' instructions and insert a JSR CHROUT instruction.

```
;read a sequential file
    LDA #addrlo
    STA datloc        ;store the LSByte of data pointer in page 0
    LDA #addrhi
    STA datloc + 1    ;store the MSByte of data pointer in page 0
    LDY #00           ;set up constant for indirect indexed data storage
    LOOP:
        ;read a byte from the input channel
            JSR CHRIN             ;get byte from input channel
      EXITIF: ;EOF condition occurs
        JSR READST                ;check for EOF
        BNE ENDLOOP
        ;store byte
            STA (datloc),Y        ;store byte in array
            INC datloc            ;set pointer to next array position
            BNE EXITIF
            INC datloc + 1        ;necessary if crossed 100h address page
ENDLOOP:                          ;program continues from here
```

Of course, a sequential-file input channel must be created with channel management code before this construct can be executed. Also, if this construct will be used to read different files into multiple memory locations, the first four instructions, which initialize the data pointer to a constant address, should be replaced with instructions that load the data pointer with the destination address for the particular data being read. The new instructions would probably be placed into modules that would call the sequential read module containing the construct above.

As with reading partial-file data, a program writes partial-file data until a stopping condition is detected. The data being written must first be obtained from memory or from an input channel. If memory is the data source, the stopping condition will be detection of byte values marking the end of the file, a record, or a field, or that a certain number of bytes have been written. If an input channel is the source, the stopping condition will usually be the EOF marker of the input file. The pseudocode for a partial-file write construct is as follows:

```
LOOP
        obtain byte
    EXITIF stopping condition occurs
        write byte
ENDLOOP
```

If memory is the data source, the 'obtain byte' step is executed with a LDA instruction and any supporting instructions to set up data addressing (see the 'store byte' instructions of the read construct). If an input channel is the data source, the 'obtain byte' step is executed by calling CHRIN or GETIN. If GETIN is used, a compound EXITIF condition such as 'EXITIF EOF OR no data available' can be used to minimize wasted time in the write loop. Other EXITIF conditions are handled as in the read loop.

The 'write byte' step is executed by calling the kernel module CHROUT with the data byte in the accumulator. If the output channel is the screen, data may move past too quickly to be read. Inserting a delay loop within the write construct, or simply holding down the CTRL key on the keyboard, will slow the display of characters to a readable rate.

The following assembly code writes sequential XASCII data obtained from memory. This code assumes that a 0 value will follow the file data in memory. 0 is handy because it sets the zero flag when it is loaded, saving a comparison operation in the EXITIF test.

```
;write a sequential file
    LDA #addrlo              ;set up page 0 data pointer
    STA datloc
    LDA #addrhi
    STA datloc + 1
    LDY #00                  ;constant used with indirect indexed op
    LOOP:
        ;get a byte from memory
          LDA (datloc),Y
    EXITIF: ;exit if end of data
        BEQ ENDLOOP          ;done if byte equals 0
        ;send byte through output channel
          JSR CHROUT         ;send byte out output channel
        ;point to next byte in memory
          INC datloc
          BNE LOOP
          INC datloc + 1
          JMP LOOP
    ENDLOOP:                 ;program continues from here
```

As before, if this construct is used to write files from different memory locations, the first four instructions must be replaced with instructions that place different addresses in 'datloc'. Such instructions would probably be placed in the modules calling a sequential write module that contains the construct above.

In either the read or write templates, to transfer fewer than 256 bytes you would probably want to change the memory addressing mode from indirect indexed to simple indexed.

Data Cassette. Cassette tape is a sequential medium, and with current cassette players data can only be recorded and played back sequentially. This limits cassette I/O methods to whole-file and partial-file sequential I/O.

The sequential-file pseudocode and constructs can be used nearly verbatim. However, to detect the difference between an EOF and an EOT (end-of-tape) stopping condition, the program can AND the value returned by READST to check bits d6 and d7 separately, and branch conditionally on each result.

Disk. A floppy disk is nearly a random-access medium, because it is almost equally easy to access any data it contains. Files can be accessed sequentially, which allows the use of whole-file and partial-file sequential I/O, but they can also be accessed out of sequence, allowing partial-file relative I/O. The disk-specific aspects of these two types of file I/O are discussed below.

Sequential Files. Sequential-file I/O uses the sequential-file pseudocode and constructs verbatim for data channels. The disk drive supports a second type of sequential channel. As we said in the discussion of channel management, command and error messages can also be exchanged with the disk drive to be used in its control. Command messages are written to the disk drive to initiate file-handling operations such as erasure, renaming, and duplication. Error messages are read from the disk to inform the program of problems with the various disk operations. Both types of messages are coded in ASCII characters. The command messages are summarized in Table 5.9 in the form in which they are transferred over the command/error channel.

The file name in the S command can represent more than one file through the use of *wildcards*. Wildcards are characters that are placed in the file name to hold the place for any legal Commodore ASCII character. When the character $ is placed at one or more positions in a name, the resulting name represents all files whose names match the non-wildcard character positions. So the name TE$T represents files TEST, TENT, TE3T, and so on.

A second type of wildcard uses the asterisk to make all character positions starting at its position wild. For example, the name PRI* represents the files PRIMARY RUN, PRINCIPLE DATES, PRIMAEVAL STATS, and so on.

Disk error messages come in the format 'message #, error name, track, block', each field being separated by a comma and the message being terminated by a carriage return. The first, third, and fourth items are numbers, while the second item is alphanumeric. Your programs must load all four items, but they can ignore the third and fourth items because they indicate position on the disk, which is a level of information we will never deal with. The first item, the error message number, contains all the information we need for our purposes.

The most important error message numbers for the kernel-level programmer are shown in Table 5.10, with the message numbers in decimal number code since that is how they arrive from the disk.

A few more esoteric and infrequent error messages are shown in the 1541 disk drive *User's Manual,* but they are never triggered in normal use.

Disk error messages have two uses. First, a program can read them in partial-file I/O loops to adjust its actions when there are disk problems. By reading the er-

TABLE 5.9 COMMAND MESSAGES

Message	Action
N<dr:><disk name><,ID> (e.g., 'N0:disk1,A1')	Erase all files, organize disk, name disk, and give it a two-character ID
N<dr:><disk name> (e.g., 'N0:disk1')	Erase all files and rename a previously organized disk
C<dr:><newfilename>= <dr:><oldfilename> (e.g., 'C0:test=0:test1')	Duplicate old file under newfilename
C<dr:><newfilename>= <dr:><oldfilename1>, <dr:><oldfilename2>, . . . (e.g., 'C0:test= 0:ta,0:tb,0:tc')	Create new file from copies of old files placed end-to-end
R<dr:><newfilename>= <oldfilename> (e.g., 'R0:test=test1')	Change name of old file to newfilename
S<dr:><filename> (e.g., 'S0:test1')	Erase file named filename
I<dr:> (e.g., 'I0:')	Return drive(s) to normal state (for drive stuck in error condition)
V<dr:> (e.g., 'V0:')	Return wasted space to usable state (not to be used with random files)
P<,lfn><,record # low-byte> <,record # high-byte> <,byte w/i record>	Select the given record and byte (i.e., field) position within the given file (for relative-file I/O)

ror messages, a program can detect the end of a relative file (discussed in the next section) or know to inform the user of simple problems such as an aborted write due to a write-protect tab on the disk. However, most of the messages indicate problems with command messages, file accesses, or channel management situations that usually do not occur once the program works properly.

The latter type of message is used as an aid in debugging. If a problem occurs with a section of partial-file I/O code, instructions can be inserted after the disk access instructions to read the error channel. Once the problem has been identified and solved, the error-channel instructions can be removed. The more complicated errors mentioned above can usually be eliminated in this way.

Command and error messages are moved to and fro through the command/error channel much as normal data are moved through a sequential-file data channel, using CHRIN and CHROUT. One of the differences between I/O through the command/error and through the data channels is the lack of a READST EOF indication after reading the last byte of an error message. The carriage return

TABLE 5.10 ERROR MESSAGES

Message no.	Meaning or probable cause
0	No error
1	A file or files have been erased (no error)
2–19	No error
21	No disk in drive, or disk is not formatted for use
25	Data written are different from data sent
26	Attempted write with disk write-protect tab covered
29	Disk not formatted properly
30	Illegal command received: syntax is incorrect
31	Illegal command received: not recognized
32	Illegal command received: longer than 58 characters
33	SAVE or LOAD requested with wildcards in file name
34	Illegal command received: no file name or no ':'
39	Illegal command received: not recognized
50	Read attempted beyond end of file
51	Too many characters sent to a relative file record
52	Attempted write to relative file would overflow disk
60	Attempted open for reading of write file
61	Attempted access of an unopened file
62	Attempted access of file not on disk
63	Attempted creation of file already on disk
64	File type does not match file type on disk
70	Requested channel or all five channels already in use
72	Disk full
73	Power-up condition (no error)
74	Used other than 0 in <dr:> portion of file name

character (0D hex) signals the end of an error message, and its detection must be used as the stopping condition for the read loop.

The following sequential-file read construct has been adapted to read the command/error channel. It includes the 'EXITIF carriage-return occurs' test and uses simple indexed addressing since less than 256 bytes will be read.

```
;read an error message and place it in memory
    LDY #$00
    LOOP:
        ;get a byte from the error channel
          JSR CHRIN
      EXITIF: ;exit if end of data
          CMP #$0D
          BEQ ENDLOOP          ;done if byte equals 0D hex
        ;store byte in memory
          STA errmsg,Y         ;put error message byte in 'errmsg' + Y
          INY
          JMP LOOP
    ENDLOOP:                   ;program continues from here
```

This construct assumes that a command/error channel has already been created using channel management code.

Writing commands to the command/error channel is the same as writing data to a sequential file, except that an extra step must be taken after a command has been sent. After each command, there must be a call to CLRCHN to signal the drive to execute the command and a call to CHKOUT to recreate the channel. The general sequential-write construct works fine if ENDLOOP is followed by calls to these two modules.

A shortcut method for writing just one command to the drive is to use the command message as the file name when creating the command/error channel. Calling OPEN sends the message to the disk, and calling CLRCHN causes it to be acted on.

Relative Files. Relative files are the most efficient way to store data elements whose position in a file can be calculated or at least estimated, and which are too numerous to fit in memory together. For instance, a large repetition data structure whose elements are ordered alphabetically by one of their fields should normally be stored in a relative file. A large mailing list structure such as that mentioned in Chapter 2 fits this description, as would a large phone directory ordered alphabetically. However, relative files should usually be used only with data that cannot be organized by their amount or order of usage.

As we said, the alphabetical mailing list data structure of Chapter 2 is a good example of this type of data structure. The top-level structure, or file, holds the entire list; each record holds a mailing address; each field in a record holds an independent element of the address; and each byte in a field holds an alphanumeric character. A mailing list is often used in two ways: to look up a single address and to look up all addresses. The simplest and safest assumption is that its individual records are equally likely to be accessed.

The time required to find a single record in a data structure like the mailing list can be shortened by up to several hundred times if the structure is stored in a relative file instead of a sequential file.

In most cases any record in a relative file of n equally accessed records can be located with no more than a handful of record accesses. The number and time of these accesses can be minimized in two ways. First, if the initial field of each record holds the record identifier, only a few bytes of each record need be loaded to select or eliminate a particular record from the search. Second, the processing overhead in using relative files can be minimized by initializing them to their full length at the time of their creation (initialization will be discussed shortly), using only newly formatted disks (see the command/error channel N command in the preceding section).

The same data structure in a sequential file requires on the average that $n/2$ or half the records be examined to find a single desired record. Further, all bytes from the beginning of the file to the desired record must be loaded during the search.

A relative file can be divided into up to 720 equal-sized records of 1 to 254

bytes each. This gives a theoretical maximum relative file size of about 180K bytes. However, the total capacity of a disk is 170K bytes, so a relative file must be limited in either the total number or size of its records to fit on an empty disk. As long as a program that manipulates relative files enforces these limitations, and other files are kept off the storage disk, a relative file will never overflow available storage. If either of these safeguards is absent, the program should read the disk error channel to check for the disk overflow condition ("52") when expanding the file.

Two channels are used to read from or write to a relative or REL file. They are the command/error channel, which is used to select the record and field to be accessed, and a relative-file data channel. As stated in the channel management section, the command/error channel should be created first and terminated last.

To move data into or out of a relative file, a program must identify the record and field it will access. It does this by activating the command/error link as both the input and output channels, and sending the P (or Position) command to the disk. This command is a series of bytes, one ASCII followed by four binary, in the form: "P, 60h OR command byte, low byte of record #, high byte of record #, byte position of field within record," where the command byte is the command value sent earlier to SETLFS to create the relative-file channel.

So the command to select the field beginning at byte 40 hex of record 118 hex, in the file with the link definition "8,8,8," would consist of the binary bytes "50 68 18 01 40" (the ASCII code for P is 50 hex). A similar command to select the beginning of the record is "50 68 18 01 01."

The program then reads the error message on the input command/error channel to detect a disk overflow or the end of the file. If it receives the first message, disk error number 52 for disk overflow, it means that a write to the selected record would overflow available disk storage. The write can be canceled and an error message sent to the user.

If the program detects the second error message, error number 50 for "record not in file," it means that the record selected is beyond the end of the file. A peculiarity of the kernel makes this the only way to test for the end of a relative file; the usual end-of-file value, the READST return value of 40 hex, is returned every time the last byte of a record is read.

Testing the error channel can be eliminated if the total length of the file is predefined and the program limits record accesses to existing records. This is the less flexible but faster and easier alternative.

The following code sends the P command to the disk drive, receives the first two bytes of the error message, and tests for disk overflow and end of file. Note that its basics are the same as sequential-file read and write code. This code assumes that the five bytes of the P command have been placed at a location called 'cmdloc'.

The output command/error channel and the data channel, in that order, must be created before this code can be executed. The code appears as follows:

```
;write the relative-file position command
    LDY #$00                    ;set index for command message fetch
```

```
        CMDLOOP:
                LDA cmdloc,Y    ;'cmdloc' holds first command byte
                JSR CHROUT      ;send byte out command/error channel
            CMDEXITIF: ;exit if all command bytes written
                INY
                CPY #$05        ;done if 5 bytes sent
                BEQ ENDCMDLOOP
                JMP CMDLOOP
        ENDCMDLOOP:             ;disk now has record/field position
                JSR CLRCHN      ;tell disk to execute the position command
                LDX #$0F        ;activate command/error link as
                JSR CHKIN       ;               the input channel

    ;read the disk error message
        IFDONE:
                JSR CHRIN
                CMP '5'         ;ignore message unless it is 50 or 52
                BNE DONE
        IFOVERFL:
                JSR CHRIN
                CMP '2'         ;if message = 52, handle overflow
                BNE IFEOF
                JSR OVERFL      ;'OVERFL' is special module elsewhere in prg
                JMP DONE
        IFEOF:
                CMP '0'         ;if message = 50, handle end-of-file
                BNE DONE
                JSR EOF         ;EOF is special module elsewhere in program
        DONE:                   ;either no overflow or end-of-file, or
                                ;done with overflow or EOF handling
```

One compare instruction can be removed from the command-write by placing the command bytes in reverse order in memory and letting the index count down to 0.

Assuming that no error has occurred, sequential-file assembly code can now read from or write to the currently selected record and field. Only the currently selected record can be read from or written to. However, all data from the selected field position to the end of the record can be accessed.

Whether reading or writing, the code should keep track of the byte position being accessed in the record if there is any possibility that access will be attempted beyond the record's end (an error condition). This is simple, since both the length of the record and the initial access position are known to the program. Thus the EXITIF test in the sequential read or write code will be based on the value in a counter, probably Y, signifying the last byte in the record. Once the last byte of a record is reached, the code must send another position command to access any more data.

Earlier we noted that relative file I/O is most efficient when the file is initialized to its largest size. Initialization is accomplished by creating the file, selecting what will be the last record in the file (this selection generates an error code 50 which should be ignored), and writing one or more bytes of filler data into that record. All

intermediate records are logged in a file data structure used by the disk drive to access the relative file, making expansion of the file within its intermediate records much faster later. If you read one of the intermediate records after initialization, you will find that it contains single "data byte" FF hex, which prints as the "pi" symbol on the screen.

The most common ways to order a relative file are by alphabetizing and by "hashing" the contents of its key field. A simple alphabetical scheme might be to assign one record to each two-letter combination in the alphabet, where the two-letter combination represents the first two letters of the key field data. This is an inefficient use of storage space since few data items will start with AA and even fewer start with ZZ. These records will probably be underused. Other records may be overfilled. However, it is a simple scheme, and $26 \times 26 = 676$ record numbers can easily be generated from alphabetical data. Variations on this or other alphabetical schemes can be tailored to a task to overcome many of these shortcomings.

A different method of relating specific data to record locations is called *hashing*. Although hashing algorithms are beyond the scope of this book, the concept is simple. A simple formula is developed to convert the values of all the bytes in a record's key field into a single record number within the file.

Any formula whose results are well distributed within the legal number of records is suitable; one common formula exclusive-ORs the key field's bytes into each other, shifting the result one bit left between each XOR. Adjustments must be made for factors like the length of the key field to produce a legal record number.

If a new record is being added to the file and the record number generated points to a record already in use, the hash code looks at succeeding records until an empty one is found. Of course, the code must ensure that the disk is not overfilled.

Similarly, if a particular record is being searched for and the hashed record number points to a record containing the wrong data, the hash code examines succeeding records until the correct record is found. In this case, the code must watch for the end of the file.

In summary, relative files are an efficient way to store large repetition data structures whose records are equally likely to be accessed and which can be searched for and selected by information in one of their fields. Relative files are accessed by creating a command/error channel, creating a data channel with the appropriate file name and format, selecting a desired record, and transferring data with that record. The last two steps are repeated as often as necessary to complete the processing task.

IEEE-488/Serial Bus. Sequential-file I/O loops for reading from and writing to nondisk IEEE-488 and serial bus devices are made the same way as the general sequential-file loops, except that the EXITIF condition is based on detecting end-of-data byte patterns or on some other criteria such as number of bytes transferred. The nature of the device dictates the appropriate condition, so study the device manual carefully.

RS-232. The IEEE-488/serial-bus comments also apply to RS-232 I/O.

Keyboard and Screen. Sequential-file I/O can be used from the keyboard by calling GETIN instead of CHRIN, and to the screen by calling CHROUT. This is discussed in the next section. The EXITIF condition for keyboard input will probably be the input of some character such as the carriage return. Channels to both the keyboard and screen must be created using channel-management code.

This completes our discussion of file I/O. As you shall see, interactive I/O is a variation on sequential file I/O to the keyboard and screen, with added user conveniences.

Interactive channels. Data are communicated through the keyboard and screen channels, otherwise known as the *interactive channels,* using kernel modules CHRIN, GETIN, and CHROUT.

Input from the keyboard can be performed with either of the two input modules; which is used depends on the processing needs of the program. As in sequential I/O, CHRIN waits for input and GETIN does not. If processing must continue whether or not the keyboard has been used, input should be via GETIN. If the program can wait for data from the keyboard, CHRIN provides line editing capabilities that can simplify the user's and the programmer's jobs.

To understand the simpler type of input, using GETIN, we must consider how the kernel obtains characters from the keyboard. The kernel operates on a cycle. Every 1/60 or 1/50 of a second, depending on which version of the Commodore 64 you have, an internal clock generates an interrupt to the microprocessor. As discussed in Chapter 3, an interrupt causes something like a module call to the kernel's interrupt handler code at the address stored in locations FFFE and FFFF hex, after the program counter and status register are saved on the stack.

The interrupt handler performs several tasks. Most of these are discussed in the section on system clocks later in this chapter. However, one of these tasks is to read the keyboard. If a key has been pressed, its ASCII code is placed into a 10-byte keyboard queue beginning at 277 hex. Each additional key press results in another byte being added to the queue, unless all 10 locations are filled. If the queue is filled, successive key presses are ignored until characters are removed from the queue.

Since queues are FIFOs, or first in-first out data structures, bytes are removed in the order that they were entered at the keyboard. A program calls GETIN, with the keyboard as input channel, to remove characters from the queue and return them in the accumulator. If the queue is empty, GETIN returns 0 in the accumulator, which can be tested for with a BNE or BEQ branch.

A special group of keys called the *function keys* is read using GETIN. These keys are located on the right side of the keyboard, and are labeled f1 through f8. Like the character keys, each returns a number when pressed. Their value is that they are not dedicated to any particular use, so a program can use each function key to select a particular program action. When a function key is pressed and GETIN returns its value, the program can test for that value and execute the appropriate code. The function key values are listed in the Commodore XASCII code chart in Appendix C.

More sophisticated keyboard input is available through CHRIN. An entire line can be entered, edited, and returned one byte at a time by calling CHRIN in a read loop like that used for sequential file input. On the first pass through the loop, the initial CHRIN call activates a built-in kernel editor.

The editor flashes the cursor on the screen and awaits keyboard input. From 1 to 88 characters can be entered, terminated by typing a carriage return. These characters are shown on the screen and may be edited with the insert/delete key, overtyping, and cursor positioning to any point in the line.

The editor provides two ways of entering characters from the keyboard. The most obvious of the two is to type a line and press the carriage return. Another entry method is possible because of the way the editor reads data.

The editor inputs whatever line the cursor is positioned on when the carriage return key is pressed. A program can provide a *menu* of lines, commands, processing options, and so on, for the user to select from with the up/down cursor key and a carriage return. This is much easier on the user than having to type a command.

With either method, when the carriage return is pressed the first call of CHRIN ends and execution returns to the program. The first character in the line is stored in the accumulator. Subsequent calls of CHRIN return characters in their order from the line, ending with the carriage return character. One more CHRIN call will start the editing process over again.

Therefore, the read loop must test for the carriage return character as its terminator. We have already seen a loop terminated with a comparison test, so we show the code without further explanation.

```
;READ A LINE FROM THE KEYBOARD

LDY #$00
LOOP:
        JSR CHRIN          ;read a byte, with editing
        STA destination,Y  ;store byte for later use
   EXITIF: ;end of line
        CMP #$0D           ;test for carriage return character
        BEQ ENDLOOP
        INY
        JMP LOOP
ENDLOOP:                   ;program continues from here
```

Byte output to the screen uses the same code as sequential-file output. Bytes are interpreted according to the Commodore ASCII character codes; some are displayed as characters and others serve as commands. In the latter category are the codes for changing between lowercase and uppercase letters, between reverse and normal letters, for the carriage return, and so on, in the Commodore XASCII chart in Appendix C.

The cursor is an important part of interactive I/O, both in CHRIN input and in the output display. CHROUT writes a character to the current cursor position

and then advances cursor position one place. The cursor position can be more freely controlled using the kernel module PLOT.

PLOT controls the cursor directly, using no input or output channels. We are discussing it here rather than under nonchannel I/O because it supports the interactive channels.

PLOT has two functions: Depending on the state of the carry flag when it is called, it either returns the X,Y screen position of the cursor in registers Y and X, respectively, or it places the cursor at an X,Y position passed in registers Y and X.

One of the ways it can support interactive I/O is in keyboard input using a menu. Suppose that there are multiple menus organized as a tree (Fig. 5.9). The input options in a menu can select one of the menu's branches, which is another menu, and so on down to the individual processing actions.

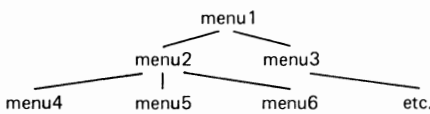


Figure 5.9

In each menu there is probably a favorite-choice option. PLOT can move the cursor to that choice at each menu level, reducing the user's work to the bare minimum. The program could even have a customization mode, where the user selects the favorite choices in each menu!

To use PLOT, you must know the X-Y numbering scheme used with the screen. It is as follows. There are 40 X, or horizontal, character positions. They are numbered from 0 for the leftmost character column to 39 for the rightmost column. There are 25 Y, or vertical, character positions. They are numbered from 0 for the topmost character row to 24 for the bottom character row. This organization can be illustrated as shown in Fig. 5.10.

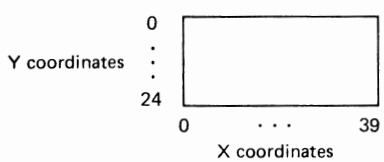


Figure 5.10 Screen

Channel I/O Summary

This completes our discussion of channel I/O. The varieties of channel I/O are summarized in the tree in Fig. 5.11. The kernel modules that perform channel I/O are illustrated in tree form in Fig. 5.12, with their normal calling order proceeding from top to bottom. Default channel selection is not shown.

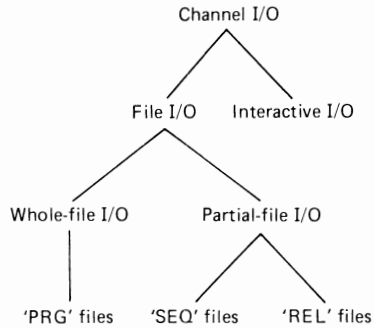


Figure 5.11

Most I/O can be performed through channels. Nevertheless, there are a few exceptions, the most obvious being the advanced graphics and audio available through the I/O chips, and the devices that attach to the game ports. Graphics and audio are discussed in upcoming chapters, but the game ports and a few remaining nonchannel I/O cases are discussed next.

USING THE I/O BLOCK: NONCHANNEL I/O

Four types of I/O work independently of channels: graphics, audio, game port, and clock I/O. The first two categories are so involved and specialized that each will be

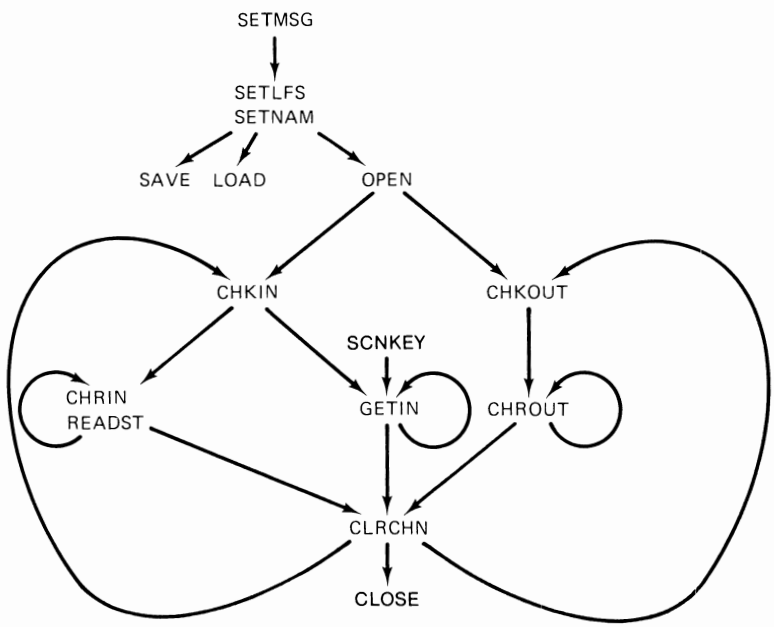


Figure 5.12

given its own chapter. The latter two categories are more closely related to general computer I/O, so they will be discussed here.

Game port devices and clocks are not supported by the kernel. Instead, I/O chips support their functions directly. To control these devices a program must directly manipulate the chip registers in the I/O block. Thus the detailed memory map of the I/O block shown earlier in this chapter will be one of your most important references for programming these devices.

Game Port I/O

There are two ports or connectors to the internal C64 buses, on the right side of the Commodore 64. They provide the connection between the C64 and light pens, game paddles, and joysticks. The connectors are labeled "control port 1" and "control port 2" on the computer, but are commonly called game ports because of the roles most of their attached devices play. Each of these devices has its own programming requirements, so they will be discussed separately.

Light pens. The light pen provides a user with a way to input data by pointing to information on the TV screen. Often the input is a selection of one program action among many, as in a menu.

Only one light pen can be used at a time. The light pen plugs into game port 1 of the C64, which is the left hand of the two game ports. Internal C64 wiring connects the light pen to the dedicated video chip, called the VIC II. As mentioned earlier, this acronym represents "the second version of the Video Interface Controller" (the first version is in the VIC-20 computer). For brevity, we will call the chip VIC.

To understand the light pen, you must understand how the TV screen "writes" a picture and how VIC can control that writing. A TV screen displays a picture by sweeping a *raster,* or electron beam, in a regular back-and-forth motion across a coating of phosphors on the inner surface of the screen. Irradiated phosphors glow for a short time.

Phosphors are grouped into small units called *pixels.* More loosely, we speak of pixels as being the smallest picture units that the computer can generate. Computer pixels are generally larger than screen pixels.

The intensity and positioning of the raster as it crosses the screen generates the detail in the picture being written. VIC does not control the raster's movement. Instead, it sends signal data to the TV screen in step with the raster, controlling the picture that the raster writes. VIC's awareness of raster position allows it to detect the position of a light pen pointed at the screen.

When the light pen is pointed at the screen and activated, VIC watches for a signal from the pen indicating that the raster has passed the pen's tip. Since VIC knows where the raster is at all times, it can record the pen's position when it receives the pen's signal. The position is recorded in two bytes, one at D013 hex, for

the horizontal position to within two pixels accuracy, and one at D014 hex, for the vertical position to the exact pixel.

The lost accuracy on the horizontal position is caused by the organization of the screen. Each VIC character is composed on an 8 × 8 pixel grid. With a horizontal by vertical screen size of 40 × 25 characters, the screen size translates to 320 × 200 pixels. The vertical number, 200 pixels, fits in a single byte, but the horizontal figure does not. The simple way used to make both numbers fit into single-byte VIC registers is to halve the horizontal resolution of the light pen.

On receiving the raster signal from the light pen, VIC will both generate an IRQ interrupt to the microprocessor and set bit 3 of location D019 if a precondition has been met: The program must have enabled the light pen interrupt by setting bit 3 of location D01A to a 1. VIC sets bit 3 of D019 on the raster signal because the microprocessor sees all IRQ interrupts the same way. The program can check bit 3 of D019 and know whether the source of the interrupt was the light pen. Interrupt routines are discussed in more detail later in the chapter.

To use a light pen from a program, include code that enables the light pen interrupt as explained above, and an interrupt handler that does two things. First, it must check D019 to verify that the light pen is the interrupt source. Second, it must read D013 and D014 to obtain the horizontal and vertical position of the pen. Interrupt handlers are discussed in greater detail later in this chapter under clock I/O. After obtaining the position values, the program can compare them against computed or stored range values and take appropriate action.

Game paddles. Game paddles provide a way to input a continuous range of values to a program. A paddle is like an automobile accelerator in that it allows relatively fine control over what value is input, but must pass over all intermediate values to proceed from one input value to another. A paddle operator, or an accelerator operator, adjusts the input by the reaction of the system being controlled. The system might be a moving object on the TV screen or the speed of the automobile being driven. If some additional feedback is provided, such as having the current input value displayed on the screen, the paddles can be used for purposes requiring more precise numeric inputs. This corresponds to having a fuel-flow-rate gauge in a car.

Most paddles are unstable enough that the value produced wavers within a range of about three values, limiting the accuracy of the input. Any binary value from 0 to FF hex can be produced by dialing a paddle from full clockwise to full counterclockwise position (Fig. 5.13). Each paddle also provides a button for simple on/off commands.

Two paddles can be used at a time. The paddles are organized into two pairs of paddles, one for each port, for a total of four paddles. Only one port can be read at a time. An easy way to discuss the two paddles on the active port is to call them L and R, for Left and Right. However, it really does not matter which paddle is physically on the left or right in any frame of reference, so do not attach any further significance to these names.

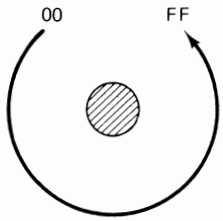


Figure 5.13

Unlike in light pen I/O, game paddles are disconnected from most of the circuits that service them until they are intentionally connected by a program. The firing buttons have the only permanent connection, by internal C64 wiring to an I/O support chip called CIA #1, the first of two complex interface adapters. It occupies locations DC00 through DC0F hex in the memory map. We will be using CIA #2 later.

Both paddles on one game port can be connected to the dedicated audio chip, called SID, by CIA #1 under program control. Recall that SID is short for Sound Interface Device. SID occupies locations D400 through D41C hex.

To use the game paddles, a program must do three things. First, it must prepare the CIA chip to make the connection between the paddles and SID. Second, it must select one of the game ports and connect it to SID. Third, it must read the data that the CIA and SID obtain from the port selected.

Preparing for Paddle I/O. When properly configured, CIA #1 can electrically attach one of the game ports to SID. Once the game port is attached, SID continuously places the input values from the port's L and R paddles into L and R game paddle registers (these registers are analog-to-digital converters).

CIA #1 is configured through its internal registers. There are two *data ports,* which are input/output registers like the configuration register on the 6510, and two *data direction registers,* which function like the register of the same name on the 6510 to control the data transfer direction of each bit in the data ports. We will call the data ports PORTA and PORTB and the corresponding data direction registers DDRA and DDRB, respectively. A 1 in a DDR bit forces the corresponding PORT bit to an output; a 0 forces it to an input.

PORTA, at location DC00, is used to select one of the two game ports for paddle I/O. PORTA's two most significant bits are assigned to this task: bit 6 to port 1 and bit 7 to port 2.

To select a game port, first disable interrupts so that the system cycle interrupt will not interfere with the paddle I/O. Second, read the value in DDRA, location DC02, and save it in memory or on the stack for later. Many kernel modules depend on the setting in this register, so you will have to restore it to its previous setting after completing paddle I/O. Third, set the top two bits of DDRA to 1's. This makes PORTA's d6 and d7 bits into outputs, which allows them to electrically attach either of the game ports to SID. At the same time the lower six DDRA bits should be 0's so that the lower bits of PORTA will be inputs. Among other things, these bits receive

game port 2's firing button inputs. So a program fixes all these bits by writing C0 hex into DDRA. The program can now select a game port.

Selecting a Paddle Port. Game port 1 or game port 2 is selected by writing a 1 into PORTA's bit 6 or bit 7, respectively. Thus the program writes 40 or 80 hex into PORTA to read from game port 1 or 2. The program must then wait a short while before reading data. A delay loop with a counter of 80 hex is considered safe by Commodore literature.

Reading Paddle Data. Two things must be read from each paddle: the numeric input value and the firing button state. The left and right paddle data can be read from locations D419 and D41A hex, respectively, on the SID chip.

The left and right firing button states are read from one of two locations, depending on which game port is currently selected. If it is game port 1, the location is PORTB (i.e., DC01 hex). For game port 2, the location is PORTA, DC00 hex. In either case, the lower seven bits of the PORT are usually all 1's. While the left paddle's firing button is pressed, bit 2 goes to 0. While the right paddle's button is pressed, bit 3 goes to 0. A program can read the PORT and test for either transaction.

PORTB is also used by the kernel for keyboard input. The kernel maintains the PORTB bits as inputs, so the paddle routine need not configure them through DDRB. However, this leads to a problem. If a program reads both the keyboard and the game paddles, a port 1 firing button press during the keyboard read will insert unwanted characters into the keyboard input. Your programs must avoid mixing keyboard input with port 1 paddle input.

Paddle reading normally alternates between selecting a game port and reading its paddles. When paddle reading is completed, the value of PORTA must be restored, and interrupts reenabled.

Code for reading both paddles on port 1 is given on the next page as a module that a program can call.

Reading both game ports requires either nearly doubling the code, or separating out a part of it as another subroutine. Try rewriting this code, with the delay through paddle reading sections as a subroutine called by the main subroutine. Convert the LDA PORTB instruction to LDA PORTA, and use an index in register X or Y with it and the STA LDATA, STA RDATA, and LDA PORTB instructions so that the code can apply to both ports.

Joysticks. Joysticks provide a way to input one of a few distinctly different values to a program. These values are arranged in two dimensions on the joystick, which corresponds to the two dimensions of the TV screen. Hence the joystick is often used to give directional commands to objects on the screen.

Another joystick characteristic is that any command can be reached from any other, without passing through intermediate values. This allows the joystick positions to be interpreted as general commands, selecting from among processing options. A close analogy is the automotive H-pattern shift. The value of the

```
;read paddles on game port 1

    porta = $DC00
    portb = $DC01
    ddra = $DC02

    ;disable interrupts
        SEI
    ;save 'ddra' for later
        LDA ddra
        PHA
    ;prepare bit directions for paddle port selection
        LDA #$C0
        STA ddra            ;makes top 2 bits of PORTA outputs
    ;select paddle port 1
        LDA #$40            ;set port 1 bit
        STA porta
        LDY #$80
     delay:
        DEY
        BNE delay
    ;read L and R paddle data
        LDA $D419           ;read the L paddle
        STA ldata           ;somewhere in memory
        LDA $D41A           ;read the R paddle
        STA rdata
    ;read paddle port 1 firing buttons
        LDA portb
        AND #%00001100      ;zero out all but the firing button bits
        EOR #%00001100      ;invert the firing button bits
        BEQ done            ;if both bits were originally 1's, neither
                            ;was pressed
        AND #$00000100      ;zero out the R button bit, leaving L's
        BEQ right           ;if L has not been pressed, R must have been
        JSR lpress          ;L has been pressed: service it
        JMP done
     right:
        JSR rpress
    ;return I/O to normal state
     done:
        PLA                 ;recover the original contents of DDRA
        STA ddra
        CLI
        RTS                 ;program continues from here
```

automobile analogy for both paddles and joysticks is that it shows the different strengths of these two types of inputs. One would not want an accelerator with only four or five positions, nor would one want to shift across 200 gear ratios to attain overdrive.

Two joysticks can be used at a time, one for each game port. Internal C64 wiring automatically connects both to the CIA. The simplicity of this shows in the joystick I/O assembly code.

The value for each of the four major axis joystick positions is assigned to its own bit in a 4-bit number. The basic "no-position" number consists of all 1's. Placing the joystick in one of the axis positions resets the corresponding bit to 0. Corner values are formed by resetting the bits for both of the two adjacent axis positions. All these values are shown in Fig. 5.14 with the joystick positions that produce them. The drawing's perspective is the view from above the joystick.

The first four bits occupy d0 through d3 of the CIA data ports. The fifth bit, d4, is dedicated to the firing button. It also resets to 0 when the firing button is pressed. PORTA, at DC00, holds the joystick data for game port 2. PORTB, at DC01, holds the data for game port 1. As pointed out in the last section, PORTB also retrieves keyboard data. Again, your programs must avoid mixing keyboard input with port 1 joystick input.

Assembly code for handling both joysticks first reads the joysticks' status from DC00 and DC01, then uses logical operations to isolate their lower five bits, and finally executes conditional branches and calls to select appropriate processing actions.

The second and last category of nonchannel I/O is clock I/O.

Clock I/O

Three kinds of clocks are accessible to your programs. The first clock is maintained and accessed by the kernel. The latter two are a clock and a timer on the CIA chips.

Kernel clock. To understand the kernel's clock, we must review the system cycle of the Commodore 64. Every 1/60 or 1/50 second, depending on the C64 model, an IRQ interrupt is generated. This is called the *system interrupt*. Its constant repetition is called the *system cycle*.

If interrupts have been enabled, this interrupt causes the processor to complete executing the current instruction, to disable interrupts by setting the I flag, and to save the contents of the program counter and the status register on the stack. The processor then retrieves the two bytes at FFFE and FFFF hex, and uses them as the address at which to begin executing.

One of the first instructions in the routine at this address is an indirect jump to the address in locations 0314 and 0315 hex. Since these are RAM locations, the address must have been placed there earlier. The kernel does this at computer power-up, with the address of a routine in the kernel.

Several functions are performed by this routine, which we earlier called an in-

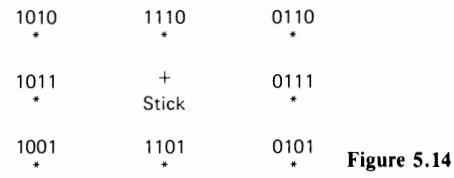


Figure 5.14

terrupt handler. The two most important to us are that it calls the kernel module SCNKEY, which reads the keyboard and places any pressed character into the keyboard queue for GETIN to remove later, and that it maintains the kernel clock. Because this is an appropriate place to do so, we will take a few paragraphs to describe the keyboard-reading function and expand on the system interrupt and cycle. Then we will focus on the kernel clock function.

Neither the system interrupt nor the kernel interrupt handler is necessary to running a program on the C64. For instance, the system interrupt can be turned off by resetting bit 0 of CIA #1's control register A, at DC0E hex, to 0. One reason to do this is that the kernel interrupt handler consumes up to about 5% of each system cycle. In a time-critical loop, the programmer may want to ensure that execution completes without a pause for interrupt handling. The interrupt can be turned back on by setting the same bit to 1. A program can do either by loading DC0E into the accumulator, using a logical AND or OR to reset or set bit 0, and writing the accumulator back into DC0E.

With the system interrupt turned off, the executing program will continue to execute normally, but without the usual periodic interruption. The kernel interrupt handler will never be activated and therefore will never read the keyboard. If keyboard input is still needed, the program must read it the same way the interrupt handler does: by calling SCNKEY periodically. SCNKEY's summary is included here for programming reference.

SCNKEY:
Purpose: Read the keyboard and if any key pressed, place its code in the keyboard queue.
JSR address: FF9F hex.
Prerequisites: None.
Data passed: None.
Data returned: None.
Altered registers: A, X, and Y.

The opposite situation, with the system interrupt turned on but the kernel interrupt handler being sidestepped, can also occur. This is done by placing the address of a routine in your program into locations 0314 and 0315 hex, which normally store the address of the kernel interrupt handler. On an interrupt, execution will pass to your routine instead of to the kernel's. This can be done two ways: First, a shortened interrupt handler may be useful to gain 2 or 3% in execution time in a critical situation which still requires frequent and regular keyboard input. Your interrupt handler might contain an interrupt counter which triggers a SCNKEY call every *n* interrupts, *n* depending on how seldomly you can tolerate a keyboard read. However, your handler must end with a CLI operation, to reenable interrupts, and an RTI, or ReTurn from Interrupt, instruction.

Alternatively, you may want all the normal interrupt handler functions together with added functions. The functions that belong in an interrupt handler are, like the keyboard read, those that require periodic and frequent processing without complicating the main program's structure or timing. Two of the most common functions in this category are graphics and audio output updating.

Adding functions to the normal interrupt handler requires inserting your own handler between the interrupt and the kernel's handler. A program does this by copying the address in 0314 and 0315 elsewhere before overwriting it with the address of its own handler. The program's handler ends with an indirect jump into the kernel's handler, using the address originally in 0314 and 0315.

Two other types of interrupts are serviced by the C64. The first is a software interrupt, or BRK. As we mentioned in Chapter 3, when the BRK instruction is fetched, the kernel routes the interrupt call to the address stored in locations 0316 and 0317 hex.

The second type of interrupt is *nonmaskable*. It is the NMI hardware interrupt, and it cannot be disabled. The programmer's main interest in this interrupt is that it is triggered when the RESTORE key is pressed on the keyboard. The interrupt call is routed to the address stored in locations 0318 and 0319 at that time.

Having discussed various keyboard reading and system cycle issues, we can return to the kernel clock. The kernel clock is a series of three bytes in low memory. The lowest byte, at 00A2 hex, is incremented by the kernel's interrupt handler on every system interrupt. The second byte, at 00A1 hex, is incremented each time the first byte overflows (from FF to 00 hex). In the same pattern, the topmost byte, at 00A0 hex, is incremented when the second byte overflows. The limit on this process for 1/60 second interrupts is the value 4F 19 FF (00A0 is on the left), which represents 24 hours. When this time is reached, all three bytes are reset to zeros to restart the clock.

There are two kernel modules for reading from and writing to this clock. However, it is so simple to read or write these bytes directly that it is not worth using the kernel. If a program using the clock must be transported to a different computer, the code can be replaced with trivial effort as long as it is well organized and well marked.

The disadvantage to using the kernel clock is that the conversion from the binary bytes to hours, minutes, and seconds, and vice versa, must be programmed, requiring much effort, much code, and much execution time. A better solution is to use the clock provided by CIA #1. It expresses time in AM/PM, hours, minutes, seconds, and tenths of seconds, and it also provides an alarm function.

CIA clock. CIA clock time is in BCD format, with each decimal digit being represented by its binary code. As with the kernel clock, the time appears to a program as a sequence of bytes in the memory map. The hours byte is kept in location DC0B hex, with the top bit indicating AM or PM time. The interpretation of that bit is up to you, but it must be used consistently within a program. The minutes byte is in DC0A, the seconds byte is in DC09, and the 1/10-seconds byte is in DC08.

The clock is set by placing the desired time into the four clock bytes, starting with the hours byte and finishing with the 1/10-seconds byte. As an example, the hexadecimal byte values for time 11:57:30.5 PM is shown below:

Hours	Minutes	Seconds	1/10 seconds
91	57	30	5

The top bit of the hours byte is set for PM, resulting in an hours value of 91.

The time must be written from hours to 1/10 seconds because a write to the hours byte stops the clock. Only a write to the 1/10-seconds byte will restart it.

A similar situation occurs when the clock is read. When the hours byte is read, the clock output is frozen, although in this case the clock itself keeps running. Only a read from the 1/10-seconds byte can release the clock output. Thus the clock must also be read starting with the hours byte and finishing with the 1/10-seconds byte.

An alarm-clock capability is included in the CIA. To set the alarm, set bit 7 of the CIA #1 control register B, at location DC0F hex, to 1. Then write the alarm time into the clock registers. Reset bit 7 of the same register to return to clock operation. When the clock time equals the alarm time, the CIA will both generate an IRQ interrupt and identify it as an alarm interrupt by setting bit 2 of CIA #1's interrupt control register, at DC0D hex, to a 1.

Note that this is the same type of interrupt as produced by the system 50 or 60 times a second. The only way a program can tell the difference is by having its own interrupt handler to determine the interrupt source. The handler must check bit 2 of DC0D hex and route processing to an alarm-servicing routine or to the kernel interrupt handler accordingly.

An alarm function is most useful with time-of-day-oriented timing. It could be used to generate periodic interrupts with a cycle of the program's choice if the program reacts to each alarm by reading the clock and placing a set increased time in the alarm registers. However, this is an awkward use of the clock. Alternatively, the clock could be used to count the time between events, by reading the clock at the beginning and ending event. However, this also is cumbersome. Both the generation of adjustable period interrupts and the timing of events are better and more easily done with the CIA timers.

CIA timers. There are two timers on each CIA chip, designated timer A and timer B. Both timers on CIA chip #1 are in use during normal kernel operations, so they are unavailable to your programs. However, both timers on CIA chip #2 are available as long as the kernel's RS-232 capabilities are not in use (i.e., as long as no channel has device number 2). If RS-232 I/O is performed through a bus converter, as we have recommended, CIA #2's timers A and B will always be available.

Each CIA timer uses two counting bytes. On CIA #2, timer A's low byte is at DD04 and its high byte is at DD05. Timer B's bytes are at DD06 and DD07.

To use a timer, a program first places a value in a *latch* at the same locations as the timer's two counter bytes. It then commands the timer to load the bytes from the latch into the counter. Next it starts the timer. Timer A decrements the 16-bit value upon receiving each 1-megahertz clock pulse (1 million times a second). Depending on its setting, timer B can decrement its counter the same way, or upon each decrement to 0 of timer A. The latter type of decrement allows using both timers together for a longer delay between timer B decrements to 0.

When a timer's count reaches 0, an IRQ interrupt is generated and a bit is set in the interrupt control register. This is the same register, although not the same bit, that the CIA flags on an alarm interrupt. As before, a program's interrupt handler can check the register and determine the interrupt's source. The CIA also reloads the starting value from the latch into the counter bytes. Depending on the timer mode commanded by the program, the timer halts or starts over again. The former possibility is called the *one-shot mode;* the latter, the *continuous mode.*

There are seven important timer registers on a CIA. Timer A and timer B each have two counter registers, at locations DD04 through DD07 on CIA #2. There is an interrupt control register at location DD0D to flag the source of the interrupt. Bit 1 is set for a timer B interrupt and bit 0 for timer A. Last, there are two command registers for selecting the different timer operations, at locations DD0E and DD0F. These operations are listed in Table 5.11.

Program code using the timers must select the timer repetition and decrement modes, place two bytes in the timer latch, command the timer to load the latch into the counter, and start the timer. The program's interrupt handler must check the CIA's interrupt control register for the interrupt source.

This completes our exploration of general channel and nonchannel I/O on the Commodore 64. In the next two chapters you will learn of the special graphics and audio effects made possible by the VIC and SID chips, and how to use them.

TABLE 5.11 TIMER OPERATIONS

Command	Register; bit; value
LOAD LATCH INTO COUNTER	
TIMER A	DD0E; bit 4; 1
TIMER B	DD0F; bit 4; 1
SELECT TIMER REPETITION	
TIMER A	DD0E; bit 3; 1 one-shot, 0 continuous
TIMER B	DD0F; bit 3; 1 one-shot, 0 continuous
START/STOP TIMER	
TIMER A	DD0E; bit 0; 1 start, 0 stop
TIMER B	DD0F; bit 0; 1 start, 0 stop
SELECT TIMER B DECREMENT	
ON 1 MHz CLOCK	DD0F; bit 5; 0
ON TIMER A TO 0	DD0F; bit 5; 1

Exercise:

Answer the following questions to review general Commodore 64 I/O.

(a) What two major types of I/O does the C64 support?

(b) Name the three types of files that the operating system allows accessing, and explain their various uses.

FOR FURTHER STUDY

Subject	Book	Publisher
C64 Disk I/O	*VIC-1541 User's Manual*	Commodore Business Machines, Inc.

AWAKENING THE *PIXY*: ADVANCED GRAPHICS

Nothing energizes a program like spectacular graphic effects. However, graphics alone cannot carry a program. For instance, many game programs include interesting scenes and motion but still bore us because their plots are flawed. To keep graphics in their proper perspective as well as to provide a solid programming background, we have delayed this chapter until late in the book. If you have enthusiastically practiced the principles of good programming, you now have control over the computer's processing actions and are ready to utilize the Commodore 64's powerful graphics to support your program tasks.

Graphics can enhance almost any type of program. The widespread belief that striking graphics are only for game programs is unfortunate: The mind can take in many types of information more quickly from a picture than from text. Further, people work better when having fun, and most people find watching graphics more enjoyable than reading text.

Nevertheless, abstract ideas are often best conveyed as text. The ideal display for many purposes combines *both* graphics and text. This is borne out by military studies on aircraft displays, which prove that the most piloting information is communicated in the least time by mixing graphics and text. Such a mixture is natural to the C64.

C64 graphics are generated by the VIC II chip. "VIC II" stands for Video Interface Controller (version) II. All graphic output, even through the screen channel as we discussed in Chapter 5, uses VIC. By using the screen channel we avoided most graphics housekeeping duties, but we also traded off most of VIC's graphics capabilities.

Now we will make use of those untapped capabilities and exploit shape, color,

and movement with VIC and the C64. The latter three characteristics are also found in cinema, the traditional art form most similar to computer graphics. We will use the cinema analogy to illuminate the use of graphics on the C64.

GRAPHICS AND THE MOVIE

Both the movie camera and movie film have equivalents in a C64 system. Only the scene to be "photographed" is left for the program and programmer to supply. VIC is the camera, obtaining the scene as data from memory and transforming it through its "lenses" to yield the screen image. We begin with the screen image since it, like film in movies, is the product of the medium.

The Video Screen as Film

The physical characteristics of the video screen and movie film are similar. Both have a "base" and an "emulsion." For movie film, the base consists of plastic or acetate, coated with an emulsion of light-sensitive particles. For the video screen, the base is the inner glass surface of the picture tube, while the emulsion consists of its regularly spaced phosphor particles. As we said in Chapter 5, the raster electron beam sweeps across the phosphors one row at a time starting from the top of the screen, taking the place of light in setting the color and brightness of the individual emulsion particles. Upon reaching the bottom of the screen, the raster beam is turned off and returned to the top row to start over. Again, in practical use the phosphors are grouped into pixels, which are the building blocks of an image. Each raster line is one pixel high.

The base and emulsion are the lowest-level picture elements. They are organized into individual pictures called *frames*. Frames are still pictures of a uniform size, which are shown in sequence to give the illusion of movement. This concept is so familiar, due if nothing else to the pictures of film reels preceding afternoon or late-night movies, that the movie parallels will be left to the reader throughout much of the following discussion.

Screen format. The first aspect of frames, that they are still pictures of a uniform size, governs the format of every C64 screen image. A full screen corresponds to a single movie frame. It contains a unicolor rectangular border surrounding a smaller *viewing screen* of two possible sizes in each dimension. The smaller screen has a height of 200 or 192 pixels, or equivalently, raster lines, and a width of 320 or 304 pixels. These sizes correspond to a screen 25 or 24 character rows tall by 40 or 38 characters wide, respectively.

The full screen is defined by 262 raster lines: 200 or 192 for the viewing screen and 62 or 70 for the border area and for the lines that could otherwise be written during the time it takes the raster beam to return from the bottom of the video screen to the top (the "vertical retrace" of the raster beam).

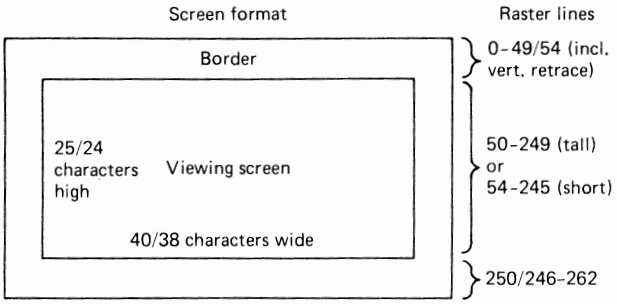


Figure 6.1 Screen format

This screen format is illustrated in Fig. 6.1. Each figure following a "/" corresponds to the smaller viewing screen size. The viewing screen size is selected by setting bits in two registers (Table 6.1).

For some purposes the viewing screen effectively extends beyond these visible pixels to lie beneath the border. With both screen sizes, this underlap allows certain picture objects to enter and exit the visible viewing area smoothly.

Additionally, with the smaller size viewing screen the entire image can enter and exit the visible area smoothly. This smooth movement of the image up and down or side to side is called "tilting" and "panning," respectively, in movie terminology, and *scrolling* in computer terminology. We will discuss the general case of scrolling the entire small-screen image here and leave the restricted case of moving individual objects for later.

Scrolling. Smooth scrolling implies that an image enters and exits the screen in small increments, much smaller, say, than the jump that occurs when the kernel editor shoves the screen up a line to insert text at the bottom.

Given that the pixel is the smallest picture unit, one pixel is the smallest amount the screen image can conceivably be moved. VIC supports one-pixel movement vertically or horizontally with two three-bit scrolling registers. The vertical scrolling register is in bits 0 through 2 of VIC's control register A at D011. The horizontal scrolling register is similarly in bits 0 through 2 of control register B at D016, beside the horizontal size selector bit d3.

The three-bit scrolling registers allow for up to eight pixels' image movement, which is sufficient for continuous smooth scrolling. Incrementing the horizontal scrolling register from 0 to 7 moves the image from left to right. This uncovers the

TABLE 6.1 SETTING THE SCREEN SIZE

Dimension	Pixels/Chars	Register location	Bit	Value
Vertical	200/25	Cntrl rgstr A (D011)	d3	1
Vertical	192/24	Cntrl rgstr A (D011)	d3	0
Horizontal	320/40	Cntrl rgstr B (D016)	d3	1
Horizontal	304/38	Cntrl rgstr B (D016)	d3	0

hidden view under the left border while concealing the right edge of the image under the right border. Upon reaching the maximum value of the scrolling register, the data from which the image is generated must be manipulated to move the entire image to the next rightward character position, new data must be written for the hidden left column, and only then can the scrolling register continue moving the image incrementally by cycling over from 0 to 7. In pseudocode this is summarized as follows:

```
'Left_To_Right_Scrolling_Algorithm'
  Set the screen to 38 columns
  LOOP
      Set the horizontal scrolling register to 0
      LOOP
          Increment the horizontal scrolling register
        EXITIF register holds 7
          Delay for desired scrolling rate
      ENDLOOP
      Move entire image right one character position
      Write the hidden column
    EXITIF done scrolling
  ENDLOOP
END 'Left_To_Right_Scrolling_Algorithm'
```

For smooth scrolling, the time consumed by the embedded delay loop must equal the time consumed in moving the image and writing the hidden column. Of course, both times can be extended for slower scrolling. A better way to control scrolling is with a *raster interrupt handler,* a special module that executes when the raster beam reaches a predefined screen line. This technique is discussed in the following animation section.

To "move entire image right one character position" you must know how the image data are organized. This is discussed in the screen memory, color memory, and bit-map memory subsections of the upcoming "Focusing the Image" section. You must move the data within these structures in fast LOOP constructs to be able to update the screen quickly enough. Because of time constraints, scrolling is usually done in character mode rather than bit-map mode, although careful programming can make the latter possible. These two modes are also discussed in the section "Focusing the Image."

To "write the hidden column," place the data directly into screen, color, and bit-map memory; CHROUT and PLOT are too slow.

Horizontal movement from right to left is the mirror image of movement from left to right. The horizontal scrolling register is cycled from 7 to 0, image data are adjusted for leftward movement, and new image data are written for the column under the right border.

Vertical movement is similar to horizontal movement except that the two possible screen sizes differ by only one character in the vertical dimension. Unlike in

the horizontal dimension, where hidden columns are available at both ends, just one hidden vertical row is available. The scrolling register can be used to place the hidden row at either the top or the bottom of the screen. A value of 0 places the hidden row at the top; a value of 7 places it at the bottom. Incrementing the vertical scrolling register from 0 to 7 scrolls the image downward. As before, the image data must be adjusted to move the entire image one character position up or down, and new data must be written to generate the oncoming image edge.

Animation. The second aspect of frames, that they are shown in sequence to give the illusion of movement, is the general case for both movies and graphics. Where the image in movies is artificial, as in cartoons, this technique is called "animation." Since all C64 images are artificial, the title *animation* also identifies moving C64 graphics. A simple subcase of computer animation is the unchanging display, which merely omits the mechanics of regular image change.

Frames in movie film are commonly shown at one of two speeds. Silent films are shown with realistic effect at 16 frames per second (fps). Sound films are shown at the higher speed of 24 fps to improve the sound quality as the film's sound track moves through the projector more rapidly.

The video raster beam fills the screen 60 times each second. This means that each raster line is present for 1/262 of 1/60 second, or 64 microseconds. Since the 6510 microprocessor operates on roughly a 1-MHz clock cycle, with each assembly instruction consuming several 1-microsecond clock pulses, it is obvious that only a handful of instructions can execute during the life of one raster line.

Changing the image on the screen once every four 1/60-second screens yields an effective change rate of 15 fps, or nearly that of silent film. This slower rate simulates motion believably while lessening the program's screen processing load by three fourths.

To implement change at this rate, we need a means of coordinating image and screen changes. VIC provides a simple solution by indicating the actual raster beam position and also by generating an interrupt when the beam reaches a preselected raster line.

Raster beam actual and interrupt positions are both accessible through nine register bits on VIC. Nine bits are necessary to represent all 262 possible raster positions. The lower eight bits are in the raster register at location D012 hex. The most-significant bit is stored in bit d7 of control register A at D011. When read, these nine bits contain the line number of the actual raster beam position, from 0 to 262 starting at the top of the screen.

When the nine bits are written to, VIC begins comparing the deposited value against the actual raster position. When the two values become equal, VIC sets raster IRQ bit d0 in the IRQ flags register at D019 to 1. This action is most useful if the raster IRQ enable bit d0 of the IRQ enable register at D01A has previously been set to 1. Then VIC will also generate an IRQ interrupt.

In this way a program can be informed when the raster beam reaches a particular raster line, which in most cases is also a pixel row. To use this information in

animation, the program must take certain preliminary actions, and several functions must be included in the program's interrupt handler. General information on interrupt handlers can be reviewed by rereading the system clock section of Chapter 5.

The program's duties for animation are to disable all nonraster interrupts, to load a pointer in locations 0314 and 0315 hex with the beginning address of the raster interrupt handler, to place the initial 9 bit raster-line comparison value into the raster and control registers at locations D012 and D011 hex, and to set up a frame counter to allow changing the image only every fourth screen.

The program disables all nonraster interrupts by writing values into interrupt mask registers on the I/O chips that generate interrupts; VIC and the CIAs. To disable all VIC IRQs except for the raster's, the value 01 must be written into VIC's IRQ enable register at location D01A hex. To disable all CIA IRQs the value 7F hex must be written to CIA #1 and#2 locations DC0D and DD0D hex. All nonraster interrupts must remain disabled until raster interrupts are no longer in regular use. VIC's interrupts are reenabled by writing the value FF hex into location D01A. They can be selectively reenabled by writing 1 values into bits d1 through d3, for the sprite/background, sprite/sprite, and light pen IRQs, respectively. CIA interrupts are reenabled by writing the value FF hex into locations DC0D and DD0D hex. More detailed information on CIA interrupts is beyond the needs of most C64 assembly-language programmers, but information on the subject is contained in the CIA chip specification in Appendix M of the Commodore 64 *Programmer's Reference Guide.*

The best raster-line comparison value for animation purposes is 250 decimal. This value represents the first line after the visible viewing screen, so it allows the longest possible time for image changes before the raster returns to the top of the viewing screen. A comparison value within the viewable area of lines 50 through 249 usually results in a flickering on the screen, which should be avoided when possible.

Screen flickering is caused by changing a screen object just as the raster is drawing it. To avoid flickering, keep two copies of the screen at different legal screen-memory or bit-map memory areas (see the screen memory and bit-map memory subsections of the upcoming "Focusing the Image" section) and display one while changing the other. By flipping back and forth between the two screens, all changes can be made off screen and this cause of flickering is eliminated.

Example:

Show an assembly language construct that could be used to prepare for using raster interrupts. Use 250d as the raster-line comparison value.

```
;prepare to use raster IRQs
  ;disable all nonraster interrupts
    LDA #$7F     ;disable CIA-chip IRQs
    STA $DC0D    ;disable CIA-chip IRQs
    STA $DD0D
    LDA #$01
    STA $D01A    ;disable nonraster VIC IRQs
```

```
;load raster IRQ vector
  LDA #rastlo   ;LSByte of IRQ handler address
  STA $0314
  LDA #rasthi   ;MSByte of IRQ handler address
  STA $0315
;load raster-line comparison value
  LDA 250
  STA $D012   ;place lowest 8 bits into raster register
  LDA $D011
  AND #$7F    ;reset ninth bit of raster-comparison value
;create frame counter
  LDA #$03
  STA frmcnt
```

Example:

Show assembly code for returning to normal system operation after a period of handling only raster IRQs.

```
;return to normal interrupt operation
  ;enable nonraster IRQs
    LDA #$FF
    STA $DC0D     ;enable CIA-chip IRQs
    STA $DD0D
    STA $D01A     ;enable nonraster VIC IRQs
```

The raster IRQ interrupt handler's duties are more numerous than the program's. First, the A, X, and Y registers must be saved on the stack.

Second, the frame counter must be maintained to allow changing the image only every fourth screen (assuming a 15-fps rate).

Third, on every fourth screen the source data for the image must be changed to simulate the desired motion. There will be more on this shortly. This part of the raster interrupt handler can also change screen colors and provide synchronization for the screen flipping discussed earlier in this section.

Fourth, the 9-bit raster-compare value must be reloaded into the raster and control registers at locations D012 and D011 hex.

Fifth, VIC must be informed that the interrupt has been serviced. This is done by writing a 1 into the raster IRQ bit of the IRQ flags register at D019. Simply reading the IRQ flags register into a CPU register and then writing the same value back will suffice.

Sixth, the A, X, and Y registers must be pulled from the stack.

Seventh and last, an RTI (ReTurn from Interrupt) or the normal system interrupt code can be executed. The latter can be done if the system 1/60-second interrupt is turned off, as described in Chapter 5, and the 1/60-second raster interrupt is used for system timing instead.

Example:

Show the assembly code for a raster IRQ handler.

```
;handle raster IRQs
rasirq:
;save CPU registers on stack
      PHA
      TXA
      PHA
      TYA
      PHA
  ;IF fourth screen-write
      LDA frmcnt
      DEC A
      ;THEN service screen
        BNE done
        ;change screen and color data as necessary
  done:
  ;load raster-line comparison value
      LDA 250
      STA $D012   ;place lowest 8 bits into raster register
      LDA $D011
      AND #$7F    ;reset ninth bit of raster-comparison value
  ;inform VIC that IRQ has been serviced
      LDA $D019
      STA $D019
  ;retrieve CPU registers from stack
      PLA
      TAY
      PLA
      TAX
      PLA
  ;return from IRQ
      RTI
```

The raster interrupt handler is the *key* to advanced graphics work. It enables a program to utilize scrolling, animation, and masking (the topic of the next section) at the fastest possible rates, which allows more graphics activity and more background processing by the program between interrupts.

Masking. Movie film allows individual frames to be subdivided through a technique called *masking*. In masking, a frame contains multiple image areas, with an independent image in each area. A common filmic example is the overlaying of a small circle containing an image over a background image that fills the rest of the frame.

The C64 supports a limited form of masking. The screen can be divided into horizontal sections having different image sources. Each section will be the full width of the screen. At assembly language processing speeds the screen can be divided into as many sections as there are raster lines, for an absolute limit of 262 sections.

Most graphics displays will use no more than two sections; one for graphics and one for text is typical, as we mentioned at the beginning of the chapter.

To mask a graphics frame, the interrupt handler used for animation is expanded with additional tasks. Beyond its previous duties it must determine which screen section the raster is in at the time of the interrupt, change the scene to be photographed for the new section, and set up the raster comparison value that will trigger the interrupt at the beginning line of the next screen section.

Determining the screen section can be done by reading the raster position and comparing it to the raster comparison values the program uses for the different screen sections. If the position equals or is one larger than (allowing for the delay before reading the value) one of the comparison values, the screen section has been positively identified. Alternatively, a simple counter can be used to keep track of progress through consecutive screen sections. Changing the scene requires the same actions as initially selecting the scene, a topic that will be discussed later.

Setting up the next raster comparison value is quite simple once the programmer has selected a value. As in animation, the value is chosen by considering the row numbers of the viewing screen inside the border. For instance, if the screen is to be divided in half, there might be interrupts on row numbers 250 and 150, for the first invisible line before the top half, and for the line halfway down the 200 visible line numbers from 50 to 249. The interrupt handler would, on the line 250 interrupt, write the hex equivalent of 150 into the raster register. On the line 150 interrupt it would write 250 into the raster register.

Whether the screen is scrolled, animated, or masked, any image on the screen will be completely composed of *picture blocks* of three possible sizes. These blocks correspond to objects in the scene VIC photographs, a scene provided by the executing program and populated with object data structures. Our next topic is this photographic subject for the C64's graphics camera.

Data as the Scene

Early in the chapter we mentioned the three major aspects to any visual scene: shape, color, and movement. Movement has already been examined, as it will be again. However, for now we will consider the scene as a frozen view. This is consistent with our definition of a frame as a still picture.

Shape and color. Shape and color are independent qualities. This is evident from the coexistence of black-and-white and color forms of movies, drawings, and other artistic media. The data making up a graphics scene are likewise separated into shape and color attributes. Shape data consist of object data structures. The constituents of color data will be discussed shortly.

TABLE 6.2 COLOR VALUES

Value (h)	Color
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow
8	Orange
9	Brown
A	Light red
B	Dark gray
C	Medium gray
D	Light green
E	Light blue
F	Light gray

Conceptually, shape is defined by the contrasting intensities of particles in and around an object. Color is defined by the hues of those particles.

In practice on the C64, there is some linkage between shape and color. An object data structure assigns one of two to four different numbers to each pixel in an object. The color data assign a color value to each of these numbers, defining pixel hue and intensity simultaneously. There are 16 possible color values, which are shown in hexadecimal form in Table 6.2.

Visual objects. Objects come in two sizes with similar structures. Every object has some multiple of eight pixels across each horizontal row, and eight or more rows from top to bottom. This structure is illustrated in Fig. 6.2. For each pixel in the object image or picture block there is one bit in the object data structure. The pixels and the bits are both ordered row by row from the upper left pixel to its lower right pixel.

So the first byte in an object data structure defines the leftmost eight pixels on

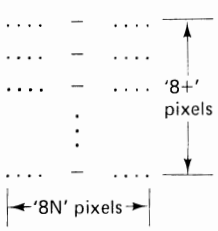


Figure 6.2 Object structure

the first row, the second byte defines the next-right eight pixels on the first row, the byte after the last byte for the first row defines the leftmost eight pixels on the second row, and so on until the rightmost eight pixels on the last row are defined. The most significant bit in a data structure byte defines the leftmost pixel in the corresponding eight-pixel object area.

VIC has three operating modes, which correspond to three different ways that VIC looks at the scene data. They are descriptively named the character, bit-map, and sprite modes and will be discussed in a later section. The object and corresponding data structure sizes for all three modes are shown in Table 6.3.

Object structure. Within each object there are two ways of correlating data bits to visible pixels. By our definition of codes (i.e., meanings assigned to low-level data elements), we can call these two correlations the *pixel codes*. VIC has *submodes* within each mode that, with just one exception, correspond to the pixel code used within the object data structures.

The simplest pixel code assigns each data structure bit to a single pixel in each visual object image. Following the left-to-right and top-to-bottom ordering convention we have already discussed, the most significant (d7) bit of the first data byte defines the value of the leftmost pixel on the top row, the next (d6) bit defines the next-right pixel on the top row, the d7 bit of the second byte defines the value of the leftmost pixel on the second row, and so on through the least-significant bit of the last byte which defines the bottom rightmost pixel.

This code is used in the normal submode of each of VIC's operating modes. So in the normal submode each pixel is assigned either a 1 or a 0 value. An object that uses this code to define the pixels in a character-mode picture block of the letter C appears as shown in Fig. 6.3. In order as hexadecimal bytes, this object contains the values "3C 7E 66 60 60 66 7E 3C."

The second pixel code assigns two bits to two-pixel horizontal groups. Following the left-to-right ordering convention, the most significant two bits in the first object byte define the leftmost two pixels on the top row, and so on.

Obviously, grouping the pixels by twos cuts the picture resolution in half. However, having two bits to describe each unit allows for assigning one of four values instead of one of two. In other words, this code trades fineness of picture detail for color variety.

The second code is the basis for the multicolor submodes of all three VIC

TABLE 6.3 VIC OPERATING MODES[a]

Mode	Object size (pixels)	Object structure size (bytes)
Bit-map	8h × 8v	8
Character	8h × 8v	8
Sprite	24h × 21v	63

[a] h and v denote horizontal and vertical pixels, respectively.

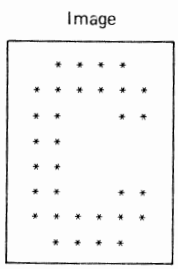


Figure 6.3

modes. Thus in the multicolor submode each two-pixel group is assigned a value from 00 to 11 binary. A "doctored" letter C in character mode and multicolor submode might appear as shown in Fig. 6.4 (with symbols representing colors). In order as hexadecimal bytes, this object contains the values "15 5A 6A 6A C0 C0 F0 BF."

Each mode has its own way of assigning color information to the data-structure bit values and thus to the pixels. These assignments are discussed in the next section.

A movie camera transforms the scene before it into images on film. We now examine the graphics device that transforms the scene data before it into screen images: the graphics camera, VIC.

VIC as the Camera

VIC has four major graphics functions, corresponding roughly to the movie camera functions of focusing, exposing film, animating, and in-camera editing. In graphics terms these functions are, respectively, transforming scene data, writing the transformed image on the screen, simulating motion, and providing for a change of scene. We have already discussed the animating and exposure functions, as well as some aspects of the focusing function. Now we will look in greater detail at the remaining focusing and editing functions.

Focusing the image. In the focusing function are included all the image acquisition and transformation actions. A lens focuses light from the scene into an image on the film. The corresponding function in VIC shares two lens attributes.

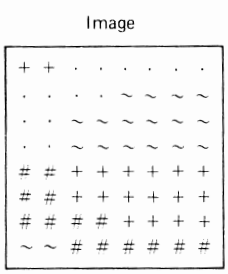


Figure 6.4

First, lenses have a *field of view.* The field of view is the area isolated from the camera's surroundings to be visible in the image. VIC also has a field of view which is smaller than its total surroundings. VIC's surroundings are the memory locations that scene data can be stored in. This includes the entire 64K memory map. VIC isolates a 16K field of view, called a *bank,* from its 64K surroundings to contain the image source. There are no overlaps between banks, so under program control VIC can access any one of up to four different banks.

VIC's field of view is selected, not on VIC, but on the data port A register of CIA #2, at location DD00h. Bits d0 and d1 select the bank as indicated in Table 6.4. Bank 0 is selected by the power-up reset handler and is active unless a program changes the selection.

A second attribute commonly associated with lenses is *filtering.* Filters are attached to lenses to transform a scene into an image in different ways. Similarly, VIC's modes and submodes treat data in the bank differently to produce different types of images. The analogy is imperfect, however, because even the basic organization of the 16K bank depends on the mode. Further discussion of the graphics functions must therefore be by individual mode.

Again, the three modes are named character, bit-map, and sprite. Only one of the first two modes can be active at a time. The third mode can coexist with either of the others, as it merely overlays movable objects on the basic image.

Character Mode. In the character mode the 16K bank contains three sections: the character sets, screen memory, and color memory.

The Character Sets. As we have already seen, in the character mode each object contains eight bytes, one byte per pixel row in the 8 × 8 pixel image. These objects are called *characters.* The *character sets* are two libraries of 256 consecutive eight-byte character definitions each. This allows a one-byte value to act as an index into the character set to select the eight-byte character to be displayed, in a way that will be explained in the following section. Thus each library requires 2K of memory, for a total character set area of 4K. The C64 provides two predefined character sets in the character set ROM, which is located between addresses D000 and DFFF hex in the memory map (when it has been placed in the memory map). The lower and upper character libraries are written in screen codes 1 and 2, respectively. These codes are shown in Appendix B.

Whatever the source, only one 2K character set can be available to VIC at a

TABLE 6.4 VIC'S FIELD OF VIEW

DD00: d1	d0	Bank	Bank range
1	1	0	0000–3FFF hex (power-up)
1	0	1	4000–7FFF hex
0	1	2	8000–BFFF hex
0	0	3	C000–FFFF hex

time. The character set in the first 2K area of character memory is selected by sending 0E hex to the screen channel with CHROUT. The second 2K library is selected by sending 8E hex to the screen.

There are two aspects of character set addressing that you must be aware of. The first is that a program can access the character set ROM only by placing it in the memory map, which locates it starting at address D000, and reading it. The second aspect of character set addressing is that because of circuitry manipulations in the C64, VIC treats the character set as if it were actually located in the memory at addresses 1000 through 1FFF hex, for VIC bank 0, or 9000 through 9FFF hex, for VIC bank 2 (see Table 6.4). This makes these address ranges unavailable for any other graphics data. Only programs and their nongraphics data can be placed in these memory areas. The character set ROM is unavailable when VIC is set to banks 1 or 3.

As we have said, the character set ROM is always present starting at a 1000 hex offset into bank 0 or bank 2 (see Table 6.4). However, the character set used by VIC can be obtained from any non-ROM bank offset that is a multiple of 800 hex. That is, VIC can be commanded to take its character set data from offset 0000, 0800, 1000, 1800, . . . , through 3800 hex into the bank.

The offset for the character set is selected by writing a value into bits d1 through d3 of VIC's bank addressing register, at D018 hex. Table 6.5 lists the bit values and resulting offsets. As usual, when altering these bits, preserve the others to avoid disturbing the other functions of the register.

Screen Memory. The data in screen memory define the character shapes on the screen. Recall that the larger screen format is 25 by 40 characters. The smaller screen is 24 by 38 characters and is created by covering rows and columns of the larger screen with a border. Screen memory is handled the same way in either case; the size of the border is selected by setting bits in the VIC registers at locations D011 and D016 hex, as was explained in the screen format section earlier in this chapter.

There is one byte in screen memory for every character position in the larger screen, for a total of 1000 bytes. The value placed in each screen-memory location serves as an index into the character set, causing the character image data for that position to be obtained from an offset into the character set equaling the byte in that

TABLE 6.5 CHARACTER SET LOCATION

D018: d3	d2	d1	Offset into bank
0	0	0	0000 hex
0	0	1	0800 hex
0	1	0	1000 hex (ROM in banks 0 and 3)
0	1	1	1800 hex
1	0	0	2000 hex
1	0	1	2800 hex
1	1	0	3000 hex
1	1	1	3800 hex (2K wrap-around)

screen-memory location times 8 (characters are defined by eight bytes of data each). Thus a value of n in a screen-memory location displays the character defined as $8 \times n$ bytes into the character set. There are 256 characters in a library, for the 256 possible index values.

Screen memory has a byte-oriented left-to-right, row-by-row structure similar to the bit-oriented structure of characters. The zeroth screen-memory byte assigns a library character to the leftmost position on the first row, the following byte defines the next-right position on the first row, the fortieth byte defines the leftmost position on the second row, and so on until the 999th byte defines the bottom rightmost character position. This structure is illustrated in Fig. 6.5, with the screen-memory bytes ordered by the screen positions they define. Again, the values placed in the screen-memory locations are indices, or pointers, into the character set selected with the bit patterns shown in Table 6.5.

Like the character sets, screen memory can be defined to start at a number of different offsets into the 16K bank. As before, the starting location is chosen through VIC's bank addressing register at D018 hex. Table 6.6 shows the bit values in bits d4 through d7 that select the different starting offsets for screen memory.

If you change the location of screen memory from the power-up value, the kernel editor (used during keyboard input with CHRIN) will not work properly until you write the most-significant byte of the new offset address into location 288 hex. For example, if the bit value written into the memory register was 1111b, selecting the screen-memory offset of 3C00h, you would write the value 3Ch into location 288h.

Color Memory. Earlier we said that "each mode has its own way of assigning color information to the data-structure bit values" that define the pixels in an object. Object shape, as we have just seen, is defined with pointer values placed in screen memory. Object color is defined with a data structure parallel to screen memory, called *color memory,* and with VIC's background-color registers.

Color memory, unlike screen memory, has a permanent location in the 64K

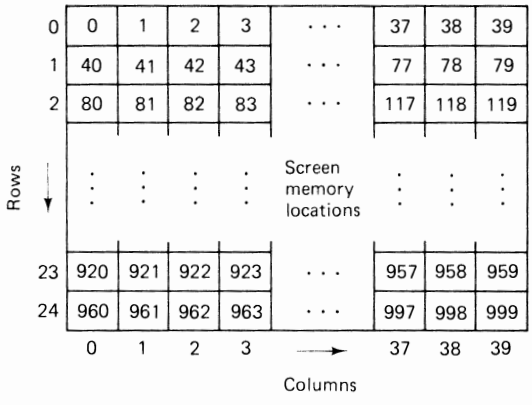


Figure 6.5 Screen memory organization

TABLE 6.6 SCREEN-MEMORY OFFSET

D018: d7	d6	d5	d4	Screen-memory offset into bank
0	0	0	0	0000 hex
0	0	0	1	0400 hex (power-up value)
0	0	1	0	0800 hex
0	0	1	1	0C00 hex
0	1	0	0	1000 hex
0	1	0	1	1400 hex
0	1	1	0	1800 hex
0	1	1	1	1C00 hex
1	0	0	0	2000 hex
1	0	0	1	2400 hex
1	0	1	0	2800 hex
1	0	1	1	2C00 hex
1	1	0	0	3000 hex
1	1	0	1	3400 hex
1	1	1	0	3800 hex
1	1	1	1	3C00 hex

memory map. It is always found starting at D800 hex. Like screen memory, color memory consists of 1000 locations that are correlated to screen positions in the same row-by-row manner. Only the lower four bits of each color memory location are used, since there are only 16 (2^4) available colors to represent. The 16-value color code used in color memory is also used in the background registers; it was shown in the "Shape and Color" subsection.

Recall that the pixels in each object are grouped by ones or twos and are represented with one- or two-bit groups in the object data structure. The size of the bit group depends on the submode. The value in each bit group is used to select the color of the corresponding pixels. Depending on the submode under which VIC is running, one of the possible bit-group values in any given screen-memory location causes the color defined by the corresponding color-memory location to be assigned to the bit-group's pixel or pixels. The color defined in color memory is the foreground color. Because there are color memory locations for every screen position, foreground color can be assigned independently to each of them. The other possible bit-group values select a background pixel color from different sources depending on the submode VIC is in. The background color for a given screen-memory bit-group value is common to the entire screen.

In Table 6.7 the color source for each possible bit-group value is shown by submode. A new submode is introduced in this table: extended background. This submode assigns the background colors in a different way that will be explained later.

Note that only eight different foreground colors can be represented on a multicolor screen without masking. This is because bit d3 in each byte of color memory determines whether the corresponding screen-memory location is multicolor or not.

TABLE 6.7 CHARACTER-MODE COLOR ASSIGNMENT

Submode	Character bit pattern	Color source
Normal	0	Background 0 (D021h)
Normal	1	Color memory
Multicolor	00	Background 0 (D021h)
Multicolor	01	Background 1 (D022h)
Multicolor	10	Background 2 (D023h)
Multicolor	11	Color memory (lower 3 bits)
Extended background	1	Color memory
Extended background	0	(See below)

The internal structure of characters in the extended background submode is the same as in the normal submode. That is, individual pixels in a character image are described by individual bits in a character data structure. Thus the same character library can be used in either submode.

However, the format of a screen-memory byte is different. Only the lower six bits serve as the index into the character library, allowing only the first 64 library characters to be used. The top two bits of a screen-memory location select one of four background colors for the character. Of course, the background color is assigned only to pixels whose data structure bits equal 0. Table 6.8 shows the background color sources for the extended background submode. The extended background submode is available only in the character mode.

The character mode is selected by placing a 0 in the Mode Select bit (d5) of VIC's control register A, at D011 hex. The normal or multicolor submodes are selected by placing a 0 or a 1, respectively, in the Submode Select bit (d4) of VIC's control register B, at D016 hex. Either of these submodes is overridden by the extended background submode when a 1 is placed in the Extended Submode Select bit (d6) of VIC's control register A. A 0 in the same location reenables the other submodes.

Bit-Map Mode. In bit-map mode the 16K bank contains three sections: bit-map memory, screen memory, and color memory.

Bit-Map Memory. In the bit-map mode the shape of the screen image is defined by the bit-map memory. In the character mode the screen image was defined by screen memory, which contained a pointer into the character set for each character position on the screen. The bit-map mode has no character sets, because it places the eight-byte character objects directly into the screen-defining data structure in place of the pointers. It follows that the resulting structure is eight times larger than screen memory, or 8K bytes long. All object bits and therefore image pixels are directly accessible in this data structure, which is why it is called "bit-map memory," or "the bit map" for short.

TABLE 6.8 EXTENDED BACKGROUND SUBMODE BACKGROUND COLORS

Screen-memory bits D7	D6	Color source
0	0	Background 0 (D021h)
0	1	Background 1 (D022h)
1	0	Background 2 (D023h)
1	1	Background 3 (D024h)

Thus the structure of bit-map memory is simply the structure of screen memory, with eight-byte character objects plugged in wherever pointers were in the previous mode. This structure is illustrated in Fig. 6.6, with each group of eight bit-map bytes in its corresponding screen character position.

To create or alter the bit-map image, you must be able to locate the corresponding byte and bit in the bit map. There are two ways of doing this, and each has advantages.

The first and most obvious way to calculate bit-map position is in terms of screen character position. Since characters are organized by row and column, bit-map positions can be calculated the same way. The equation for locating the bit-map position for any pixel on the screen is:

bit-map byte location
= (character row × 320) + (character column × 8) + pixel line

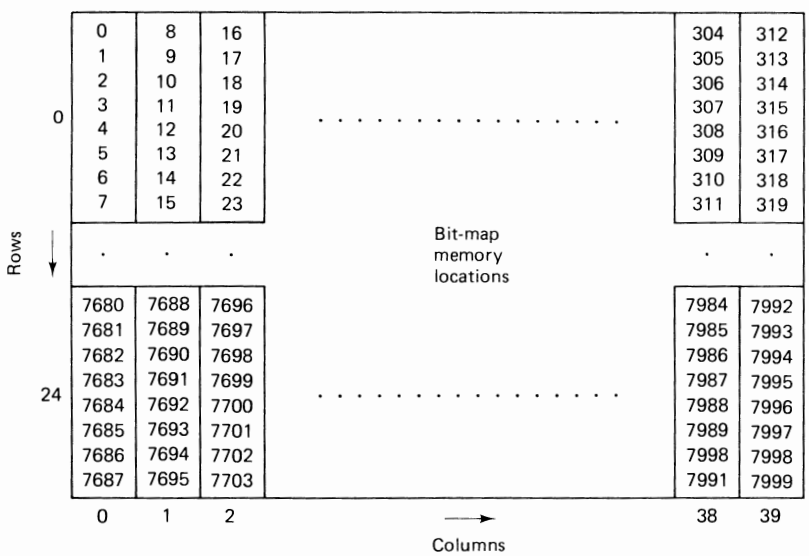


Figure 6.6 Bit-map memory organization

where character row ranges from 0 to 24, character column from 0 to 39, and pixel line from 0 to 7.

The bit location for a particular pixel is the same as the pixel position on the pixel line; that is, the leftmost pixel on the pixel line is represented by the leftmost bit (d7) of the byte, and so on. To ensure that you know how to use the equation, try calculating the bit-map location for a pixel row in any character shown in Fig. 6.6.

This method of calculating data structure position is particularly useful when the screen is masked into a combination of bit-map and character mode sections. Using the equation above, the entire screen can be designed and handled in terms of character position.

A second method of correlating data structure and pixel position is in terms of pixels. The screen is divided into 320 pixel rows and 200 pixel columns. The byte and bit location for the pixel at any X,Y column and row position is calculated with an equation derived from the previous character-based equation. This equation uses two arithmetic operations called DIV and MOD to translate pixel row and pixel column values into character row, character column, and pixel line values. DIV produces the quotient of a division operation while discarding the remainder. MOD produces the remainder of a division and discards the quotient. So DIV(13/8) produces the value 1, and MOD(13/8) produces 5. Both DIV and MOD can be programmed from variations on the division algorithm of Chapter 3. The adaptation is simple and is left to you as an exercise.

The second form of the bit-map equation is

$$\begin{aligned}\text{bit-map byte location} = {} & [\text{DIV}(\text{pixel row}/8) \times 320] \\ & + [\text{DIV}(\text{pixel column}/8) \times 8] \\ & + [\text{MOD}(\text{pixel row}/8)]\end{aligned}$$

Again, the bit position for a particular pixel is calculated from left to right in eight-bit groups. MOD can be used to calculate the bit position from the pixel column. The relationship is

$$\text{bit position} = 7 - \text{MOD}(\text{pixel column}/8)$$

So the bit position for a pixel in the tenth pixel column is 7 – MOD(10/8), or 5, which represents bit d5 in the bit-map byte location.

The DIV and MOD operations in these equations are applied to divisions by 8. Both operations can be performed at once by shifting the variability quantity, either pixel row or pixel column, three bit positions right and catching the 3 bits that fall off. The quantity remaining from the original variable is the DIV result and the three dropped bits form the MOD result.

Example:

Illustrate DIV and MOD using bit shifting on the problem 13/8.

The initial contents of the Store and Catch bytes are

Store	Catch
0000 1101	000

The contents of Store and Catch after the bit shift are

Store	Catch
0000 0001	101

In the example the DIV result of 1 is in the Store, and the MOD result of 5 can be read from the Catch. In practice, it is faster to make two copies of the variable, shift one copy right 3 bits to obtain the DIV, and isolate the lowest three bits of the other copy with an AND 0000 0111 operation to obtain the MOD.

The pixel-position equations treat the screen as a single large object. Individual pixels can be manipulated according to their X,Y position on the screen, without regard for character position. This view corresponds to the normal usage of the bit-map mode and is convenient for that purpose. It also facilitates graphic applications that plot pixels by their Cartesian (X,Y) or other coordinate system location.

As with the screen memory and character mode, bit-map memory can be placed in more than one position within VIC's address bank. Bit d3 of the bank addressing register at D018 hex selects this position. The two possible locations are shown in Table 6.9.

There are two submodes in the bit-map mode, normal and multicolor, which correlate single or paired bits in each character object to single or paired pixels in each character position on the screen. The method of bit/pixel correlation for both submodes is described in the internal structures subsection of the section "Screen as Film."

With image shape fully described by the bit map, color is the only remaining image quality to be defined. In the bit-map mode, this definition is provided by screen and color memory.

Screen and Color Memory. Screen and color memory are organized the same way as in the character mode: each section's 1000 locations correspond to the 1000 character positions on the video screen in a left-to-right, then row-by-row manner. So one screen-memory byte and one color-memory nibble are dedicated to each eight-byte object in the bit map.

TABLE 6.9 BIT-MAP LOCATION

D018: d3	Offset into bank
0	0000 hex
1	8000 hex

TABLE 6.10 BIT-MAP-MODE COLOR ASSIGNMENT

Submode	Character bit pattern	Color source
Normal	0	Screen memory low nibble
Normal	1	Screen memory high nibble
Multicolor	00	Background 0 (D021h)
Multicolor	01	Screen memory high nibble
Multicolor	10	Screen memory low nibble
Multicolor	11	Color memory nibble

Like the color memory nibble, in the bit-map mode each screen memory nibble assigns a color value to a particular bit-pattern value in the eight corresponding object bytes. VIC's background-color register 0 is also used as a color source in one submode. The color source for each object bit pattern in each submode is shown in Table 6.10.

Note the extra color flexibility of the bit map's multi color submode over that of the character mode. By using screen memory instead of background registers for two of the color values, three colors are independently selectable for each character position instead of just one. Because there are no character pointers in the bit-map mode, there is no extended background submode.

The bit-map mode is selected by placing a 1 in the Mode Select bit (d5) of VIC's control register A, at D011 hex. As in the character mode, the normal or multicolor submodes are selected by placing a 0 or a 1, respectively, in the Submode Select bit (d4) of VIC's control register B, at D016 hex.

Sprite Mode. In the last graphics mode, small and mobile objects traverse the background provided by either of the other modes. These objects are called *sprites,* for the fairy creatures of magical powers and benevolent though impish intentions.

Each sprite is 24 pixels wide by 21 pixels tall. Thus there are three bytes per row and 21 rows in a sprite, requiring 63 bytes in the object data structure that assigns numerical values to pixels. Object bytes correspond to image pixels in the same way as in the character mode. This structure is illustrated in Fig. 6.7. Data structure bytes are numbered within the pixel borders.

Bits are assigned to pixels in the usual left-to-right manner. Normal and multicolor submodes exist and follow the same bit and pixel grouping rules as in the character and bit-map modes (see the discussion in the character mode section). The color assignments for both submodes will be shown later.

Up to eight sprites can be displayed at once on an unmasked screen. Their data structures can be located anywhere in the 16K bank and need not even be grouped together because the location of each structure is independently defined. VIC assumes that eight one-byte pointers, one for each sprite, are located in the last eight bytes of the bank's 1K screen memory section. That is, if screen memory is located at 0400 hex, screen memory fills the first 1000 bytes of the 1024 in the section. This takes

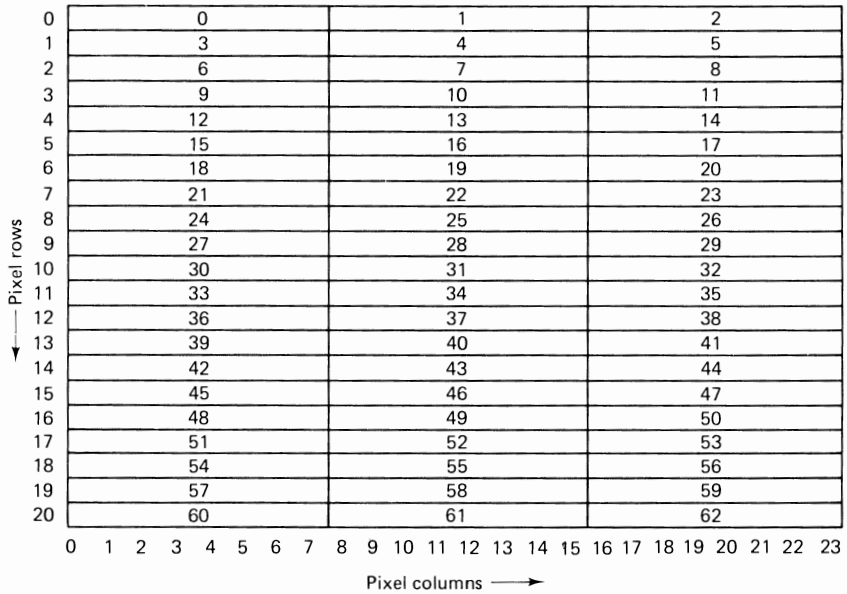


Figure 6.7 Sprite organization

up bytes 0400 through 04E5. The sprite pointers fill the last eight bytes of the 1024, from 04F8 to 04FF.

The pointers are one byte long to allow dividing the 16K bank into 256 64-byte segments. Each segment just holds a 63-byte sprite data structure, with a leftover byte at the end. The actual data offset is the product of the pointer value and the number 64:

$$\text{sprite-data offset into bank} = \text{pointer} \times 64$$

So a pointer value of 0 tells VIC that the data structure is at the beginning of the 16K bank, and so on.

Although only eight sprites can be displayed on an undivided screen, by using masking additional sprites can be displayed. For instance, multiples of eight sprites can be displayed on the same background image by changing the sprite pointer values on the interrupts for multiple screen sections. Of course, sprite data must exist for each different pointer value.

Fewer than eight sprites can be displayed in a screen section or on an unmasked screen by fixing bits in VIC's sprite enable register at D015 hex. Each of the eight sprites has its own enable bit. To turn on sprite *n*, set the *n*th bit in D015 to 1. So, to turn on sprite 7, set bit d7 to 1. To turn off a sprite, reset its bit to 0.

Each sprite image can be doubled in size vertically, horizontally, or both by setting bits in VIC's sprite expansion registers. To expand sprite *n* horizontally, set

the *n*th bit in VIC's horizontal expansion register at D01D hex to 1. That is, set bit d0 for sprite 0, bit d1 for sprite 1, and so on. Reset the bit to 0 to return the sprite to normal size. To expand a sprite vertically, set the corresponding bit in VIC's vertical expansion register at D017 hex to 1. Again, the bit positions correspond to the sprite numbers.

A sprite's screen position is the X,Y location of its upper-left corner. Each sprite image can be placed at any X,Y coordinate on the visible screen. Additionally, sprite images can be hidden under any part of the screen border. To accommodate all these positions, the available number of horizontal and vertical pixel positions has been expanded from 320 and 200 to 512 and 256.

The new position ranges include 192 hidden positions horizontally and 56 vertically. The visible area is skewed within the overall ranges, so the hidden border positions on each side of the viewing screen are of different sizes. However, the position values wrap around from the highest to the lowest numbers, connecting the two borders on either side of the viewing screen. By taking advantage of this, a program can move a sprite smoothly onto or off the visible screen in any direction. For instance, a sprite that is carried off the right side of the screen by incrementing the X-position value will reenter the screen from the left as the X-position value is incremented through 512 to 0 and on up again.

The coordinates for hidden, partially visible, and visible sprites of both sizes on both size viewing screens are given in Table 6.11. Where a range specifies a larger position value followed by a smaller (e.g., 250–28), a wrap-around from the highest to lowest values is included.

TABLE 6.11 SPRITE POSITION COORDINATES

Screen/sprite	X coordinates (0–511)	Y coordinates (0–255)
320 × 200		
Standard sprite	Hidden: 344–0 Left partial: 1–23 Visible: 24–320 Right partial: 321–343	Hidden: 250–29 Top partial: 30–49 Visible: 50–229 Bottom partial: 230–249
Enlarged sprite	Hidden: 344–488 Left partial: 489–23 Visible: 24–296 Right partial: 297–343	Hidden: 250–8 Top partial: 9–49 Visible: 50–208 Bottom partial: 209–249
304 × 192		
Standard sprite	Hidden: 335–7 Left partial: 8–30 Visible: 31–311 Right partial: 312–334	Hidden: 246–33 Top partial: 34–53 Visible: 54–225 Bottom partial: 226–245
Enlarged sprite	Hidden: 335–480 Left partial: 481–30 Visible: 31–287 Right partial: 288–334	Hidden: 246–12 Top partial: 13–53 Visible: 54–204 Bottom partial: 205–245

TABLE 6.12 SPRITE-POSITION REGISTERS

Address	Function
D000	Sprite 0 X coordinate (least significant byte)
D001	Sprite 0 Y coordinate
D002-D003	Sprite 1 X and Y coordinates
D004-D005	Sprite 2 X and Y coordinates
D006-D007	Sprite 3 X and Y coordinates
D008-D009	Sprite 4 X and Y coordinates
D00A-D00B	Sprite 5 X and Y coordinates
D00C-D00D	Sprite 6 X and Y coordinates
D00E-D00F	Sprite 7 X and Y coordinates
D010	Most significant bits of the 8 X coordinates

The limiting coordinates 512 and 256 were chosen for practical reasons. There are 256 positions maximum that can be represented with eight bits, and 512 maximum with nine bits. VIC supplies eight bits for specifying the vertical position and nine bits for specifying the horizontal position of each sprite. These values are stored in the sprite position registers, from D000 to D010 hex. The first 16 locations are paired into eight combinations of horizontal and vertical coordinates for the eight sprites, starting with sprite 0 at location D000 hex. Each horizontal coordinate register omits the ninth or top bit, which is placed in the seventeenth register at D010 in the *n*th bit position for the *n*th sprite. That is, bit d0 of D010 holds the most-significant bit of sprite 0's horizontal coordinate. The sprite position registers are represented in Table 6.12.

Colors are assigned to sprite bit values according to the submode. Eight nibble-wide sprite color registers, from D027 through D02E hex, store a dedicated color value for each sprite.

The only unusual thing about color and sprites is that in both submodes one bit pattern allows the underlying image to show through. In effect, the corresponding sprite pixels become transparent. The color assignments for both submodes are given in Table 6.13.

A final characteristic of sprites is that they interract with each other and with the underlying image. This characteristic has two consequences. First, sprites have priorities for line-of-sight order. This facility simulates three dimensions. A sprite of

TABLE 6.13 SPRITE-MODE COLOR ASSIGNMENT

Submode	Sprite bit pattern	Color source
Normal	0	Underlying image
Normal	1	Sprite color register
Multicolor	00	Underlying image
Multicolor	01	Multicolor register 0 (D025)
Multicolor	10	Sprite color register
Multicolor	11	Multicolor register 1 (D026)

a lower number will always cover a sprite of a higher number (e.g., if sprite 2 and sprite 7 cross, sprite 2 will show and only those parts of sprite 7 under sprite 2's transparent pixels will be visible). Each sprite can also be made to pass over or behind the underlying mode image. By resetting the *n*th bit of VIC's display priority register at D01B hex to 0, the nontransparent pixels of sprite *n* will cover the image from the underlying mode. By setting the same bit to 1, the sprite's nontransparent pixels will show only above underlying image areas defined by background color register 0, or defined by multicolor bit pairs assigned the value 01.

Second, sprites can collide with each other and with the underlying image. All 0-bits and 00 and 01 bit pairs in an object of any mode are considered to be background areas. All other bit or bit-pair combinations are considered to be foreground areas. A collision is registered only when a foreground bit or bit pair of a sprite crosses over a foreground bit or bit pair of another sprite or of the underlying image.

A collision has three effects. First, VIC indicates which sprites have collided by setting 1-bits in one of two collision registers. The latter registers correspond to the two types of collisions; there is a sprite/sprite register at D01E for sprite-to-sprite collisions and a sprite/image register at D01F for sprite-to-image collisions. In a now-familiar pattern, the *n*th bit of each register represents sprite *n*.

The second effect of collisions is that an IRQ interrupt is generated. The third effect is that VIC indicates the cause of the interrupt by setting one of two bits in the IRQ flags register at D019. Bit d1 indicates a sprite-to-image collision interrupt, and bit d2 indicates a sprite-to-sprite interrupt.

As before, the IRQ flags register must be cleared after the interrupt, which we have seen done by reading the register and writing back its contents. The set bits in the collision registers are cleared automatically when the registers are read from.

Sprite-to-sprite collisions can always occur off-screen. Sprite-to-image collisions can occur off-screen in the horizontal direction if the smaller screen size is selected and the image has been scrolled under the border.

The sprite mode is always selected. Whether or not sprites are displayed depends on whether sprites have been enabled, placed in the visible section of the screen, and given nontransparent colors.

Each sprite's submode is independent. Thus any mixture of normal and multicolor sprites is legal. Sprite submode is selected by altering the appropriate bit in VIC's sprite submode register at D01C hex. A 1 in the *n*th bit of D01C places sprite *n* in the multicolor submode. A 0 places the sprite in the normal submode.

Editing the take. VIC's last cameralike function is to handle transitions between scenes. Transitions are uncomplicated but dramatic effects whose potential is often ignored in computer graphics. Most other camera functions are static, but transitions are dynamic and create a visual rhythm.

Two types of edits are important to this discussion: cuts and wipes. *Cuts* are abrupt transitions between scenes. The same effect can be achieved in a graphics mode by switching VIC between 16K banks. In character mode a single 16K bank has

room for multiple screen-memory sections, which can be exchanged for the same effect.

Wipes are transitions where a sharp border between the old and new images covers the screen in a pattern, usually until the new image has completely replaced the old. This is not an "in-camera" technique in movies, but it is an "in-VIC" technique in graphics, which is why it is included here.

A simple wipe would begin with using the mask technique to divide the screen into two sections. The first raster comparison value is at the top of the visible screen. On each interrupt the raster comparison value is incremented by one character line, which is equivalent to eight raster lines. At 60 frames per second, the 25 character-position changes will produce a 1/2-second wipe.

More ornate wipes are elaborations on the simple wipe. Combining the techniques in this chapter can produce many other novel effects. However, all these techniques can be used more effectively in the larger context of sight and sound. Chapter 7 explores the sound capabilities of the C64.

Exercise:

Review the major points of this chapter with the following questions.

(a) Name the C64 graphics modes and describe their similarities and differences.

(b) Name the C64 submodes and describe how each encodes the pixel bits within visual objects.

FOR FURTHER STUDY

Subject	Book	Publisher
C64 graphics circuitry	*The Commodore 64 Programmer's Reference Guide*	Howard W. Sams & Co. and Commodore Business Machines Inc.
Mathematically generated natural graphics	*The Fractal Geometry of Nature*	W. H. Freeman and Company

VOCAL CHORDS FOR A CHIMERA: SOUND SYNTHESIS

Like the chimera, the Commodore 64 is a being of varied parts. Instead of owing its lineage to the goat, lion, and dragon, however, the C64 counts among its relatives the simple computer, the graphics generator, the I/O handler, and the sound synthesizer. Of these, the C64's sound synthesizer element has the most distinguished pedigree.

For chimeras or computers an expressive voice is an emblem of power. The C64's dedicated audio chip, SID, for Sound Interface Device, provides the C64 with three of the most expressive voices in any personal computer. To control these voices, the programmer must first understand the nature of sound. He or she must then abandon the unfortunate but pervasive emphasis on specialization and take on the gestalt of the composer. Sound is a medium for communication, and the programmer can judge the sound only by its effectiveness at communicating rationally and emotionally. Any such communication can and should be musical in the broadest sense; it should be aesthetically controlled even if it is not recognizably melodic, harmonic, or rhythmic.

In this chapter we begin by studying the nature of sound. This will prepare you to use SID in a more purposeful way from your programs. We will then examine the programming requirements for utilizing SID's sound capabilities.

WAVE ATTRIBUTES

The study of the nature of sound is called *acoustics*. Acoustics is a broad field with many applications and principles beyond the scope of this discussion. We consider

the most basic concepts, and will be repayed with ample technique for creating new sounds.

Sound occurs when an object moves in a fluid such as air (we will ignore sound in solids). This movement presses adjoining molecules in the fluid together, raising their pressure above their surroundings. The compressed molecules then expand into the lower-pressure areas around them, compressing the surrounding molecules, which then expand, and so on, in an increasing radius around the moving object. This mechanism for transmitting energy is called a *compression wave,* for obvious reasons. If there are no impediments to the wave, the farthest compressed molecules will always form a sphere.

Waves have four major attributes: *intensity, frequency, waveform,* and *phase.* In human terms the first two attributes are called *loudness* and *pitch.* The third attribute is often called *timbre* or *tone color,* and can be derived from the first two. The fourth attribute is recognized by the human ear only under special and usually artificial circumstances; hence it has no common name. The modification of these attributes is called *modulation.* We will discuss the four attributes individually, and then explore the techniques provided by SID for modulating them.

Intensity

Sound waves carry energy outward from a moving object. To define how much energy is in a sound, we must place certain limits on our measurement. If we measure the energy produced by a regularly moving object over an indefinite time, the resulting value will increase without limit. For a more meaningful measurement we can imagine a sphere centered closely around a cyclically moving object, and measure all the energy passing through the sphere in 1 second. This measurement reveals the total energy produced per unit time by the object. Energy per unit time is called *power.*

If we measure the power through a single unit-area piece of the sphere, the resulting value is the power per unit area, or *intensity,* of the wave in that direction. The intensity measurement accounts for the inevitable differences in the amount of sound energy passed in different directions, and for the energy in the small wave area entering the ear.

The range of audible intensities is enormous. The loudest-bearable sound carries approximately 1 trillion times more energy per unit time per unit area than the softest detectable sound. To accommodate this range, the ear collapses perceived sound intensity into a smaller, logarithmic loudness scale. A single unit on this scale is called the *decibel,* abbreviated dB.

Only 160 decibels separate the quietest audible sound from an instantly damaging sound. The formula for calculating the decibel loudness from the actual sound intensity I and the softest audible sound intensity I_0 is

$$\text{loudness (dB)} = 10 \times \log_{10}(I/I_0)$$

This scale implies, for instance, that perceived loudness doubles with a squaring of the energy intensity ratio, rather than with a simple doubling in energy intensity. Loudness levels for some familiar sounds are listed in Table 7.1.

The loudness span for SID's output is 48 dB, from no output to full power. However, the TV or stereo set receiving this output amplifies (i.e., multiplies) it by some factor before dispersing the sound. The loudness range heard from a loudspeaker can therefore be either expanded or contracted from SID's 48 dB.

Frequency

For an object vibrating with a regular, repeated pattern, a given point in the fluid will be cyclically compressed and expanded. The time between compressions in this cycle is called the *period.* The number of periods or cycles per second is called the *frequency.* Representing the period with the variable T and the frequency with the variable f, let us express this relationship as

$$f = 1/T$$

Cycles per second have units of hertz or simply Hz, after the German physicist Heinrich Rudolf Hertz who researched waves in the late 1880s. 1000 Hz are represented as 1 kilohertz or 1 kHz.

Assuming that the speed of sound is the same for all audible frequencies in air (an assumption that is nearly true), the period of the wave will be proportional to the distance between compression points on a line outward from the sound source. The shorter this distance is at the constant speed of sound, the shorter the wave's period and the higher its frequency will be.

Air behaves like a spring. In compression, the molecules resist with increasing force as they are pressed closer. In expansion, the surrounding molecules press inward and attempt to return the expanded region to the surrounding or ambient pressure. Thus ambient air corresponds to a spring in its neutral, undisturbed posi-

TABLE 7.1 VARIOUS LOUDNESS LEVELS

Sound	Loudness (dB)
Pin dropping	0
Leaves rustling	10
Quiet room	20
Unoccupied kitchen	40
Normal conversation	60
City street	70
Garbage disposal	90
Jet 1/3 mile away	110
Jackhammer	120
Loud rock band/pain	140
Immediate ear damage	160

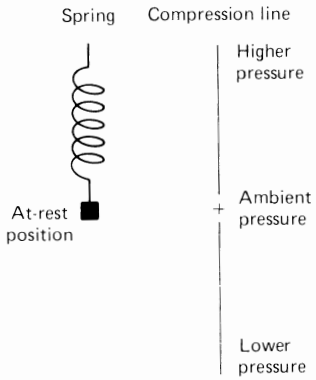


Figure 7.1

tion. We can show the motion of this spring as it is compressed and expanded by drawing a compression line beside it, as shown in Fig. 7.1.

Since the compression line is vertical, we will call the position on the compression line y (as in an x,y axis). The ambient pressure point can be defined as $y = 0$, with the highest pressure as $y = 1$ and the lowest pressure as $y = -1$.

The solution for the spring position y as it oscillates in time is

$$y = \sin(2\pi f)$$

Since the spring represents air, this is also the solution for the pressure at a physical point due to the compression wave from the simple cyclical motion of an object in air. Recall that f is the frequency of the disturbance from the moving object that causes the sound.

The $\sin(2\pi f)$ function is called the *sine function* and is one of the most basic relationships in nature. We can graph this variance of pressure with time at a point in space. The axes are built by moving the compression line to the left, and by creating a horizontal time line through the ambient-pressure point. When plotted this function appears as shown in Fig. 7.2.

Because air behaves like a spring, it can carry sound energy only as sine waves. Therefore, all sounds from the simplest to the most complex must consist of some

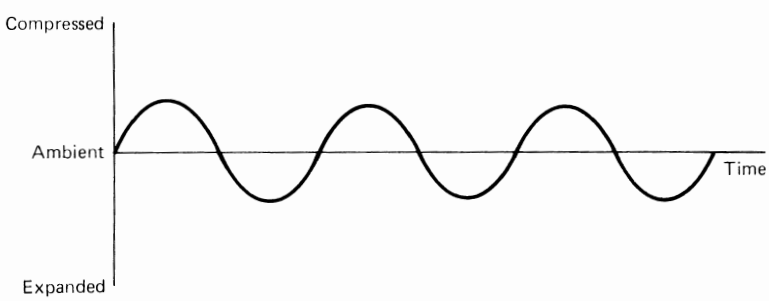


Figure 7.2 Graph three complete sine cycles, starting at ambient point

combination of basic sine waves of different frequencies and phases (a wave attribute that will be discussed shortly).

The simplest sound consists of a single sine-wave frequency. Assuming that the frequency is audible to the human ear, a simple sine wave has the clearest pitch possible. Indeed, listening to a pure sine wave quickly becomes monotonous and even irritating. We call the pure sine wave the *fundamental* frequency or pitch.

The most complex sound consists of an infinite number of sine waves covering the entire range of audible frequencies. This sound, called *white noise,* resembles rushing wind.

Between the two extremes are finite combinations of audible sine waves. Nature favors certain frequency relationships because of the way objects vibrate. A vibrating string provides a good illustration of how objects vibrate.

Consider a string that is attached at both ends to unmoving objects (Fig. 7.3). When the string is plucked, its vibration divides it into evenly spaced alternating points of maximum vibration and stillness, with still points at either end. There will always be an integer number of maximum-vibration points, which are called *nodes.* The vibration patterns for the first several integer number of nodes are shown in Fig. 7.4. In each case, for m nodes there are $m + 1$ still points (including the endpoints). The still points between nodes act like endpoints; that is, m nodes divide the string into m substrings, each $1/m$ times the length of the full string. Each substring has a wavelength of $1/m$ of the string wavelength, which according to our earlier discussion yields a substring frequency of m times that of the full string.

When a string is plucked, it vibrates with all numbers of nodes from 1 to 20 or more, as if there were 20 or more separate strings each vibrating with just one node pattern and associated frequency. The one-node pattern has the lowest frequency, of course, and is called the *fundamental,* as in the frequency of the basic sine wave. This is the frequency we call the *pitch* of the sound. The two-node frequency is twice that of the fundamental, and so on. The frequencies for m greater than or equal to 2 are called *harmonics.* Harmonics are almost always present in natural sounds, and with training can often be heard as separate pitches.

The human ear is at best sensitive to frequencies from 15 to 20,000 Hz. It perceives pitch change at the rate of the base 2 exponent of frequency change. So a frequency change of 2^1 (i.e., a doubling of frequency) results in a perceived pitch change of 1, called an *octave.* The octave is the basic *interval* or separation between pitches. Two pitches an octave apart seem to have the same pitch quality at different heights. Since the number 15 can be doubled only about 10 times in the range of human hearing, the ear has a range of about 10 octaves.

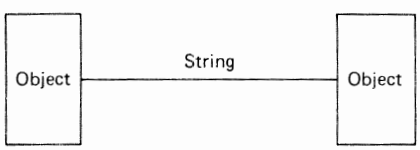


Figure 7.3

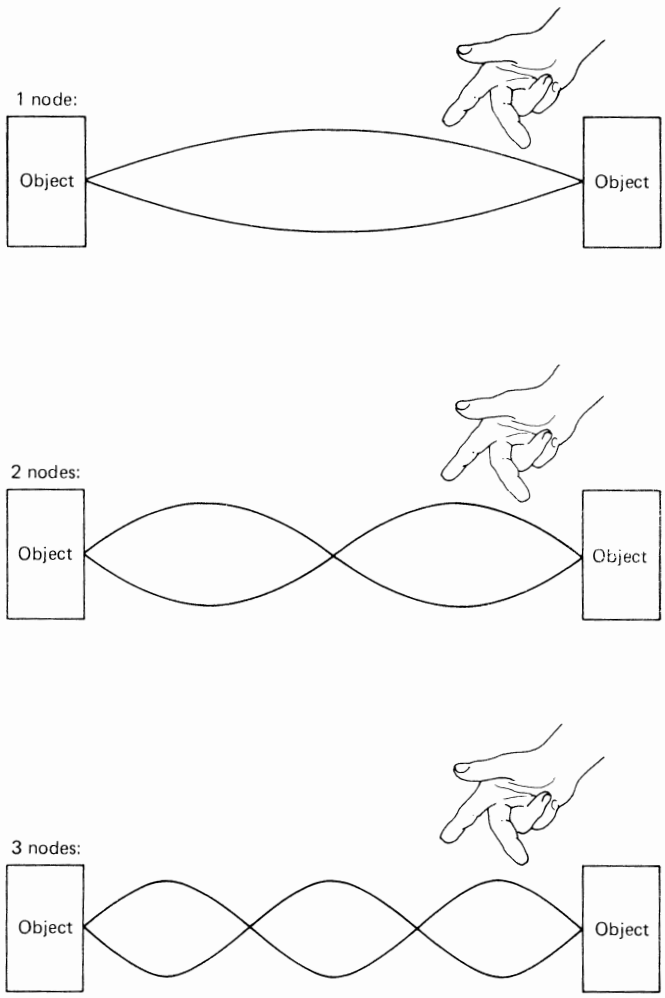


Figure 7.4

A *scale* is a subdivision of an octave into (usually) five or more intermediate pitches. Many different scales have evolved in different cultures and times. The two most important to Western music are the just and tempered scales.

The *just scale* consists of 20 unique pitches or *semitones* in each octave, derived from the harmonic series. The foundation intervals of the octave, fifth, and third are shown in Table 7.2 with their corresponding harmonic and node counts, and the ratios between their frequency and the fundamental pitch. The simple interval of the third is defined as the separation between the third and fourth harmonics, which has the frequency ratio to the fundamental of [(5/1)/(4/1)], or 5/4. Pro-

TABLE 7.2 FOUNDATION INTERVALS OF THE JUST SCALE

Nodes	Harmonic	Frequency ratio	Interval
2	First	2/1	Octave
3	Second	3/1	Twelfth (fifth, one octave up)
4	Third	4/1	Second octave
5	Fourth	5/1	Third, two octaves above

ceeding up the harmonic series from the fundamental, and making similar ratios for other harmonics, yields the just scale.

This method divides the octave unevenly. When chords are built of the octave, third, and fifth intervals, the most natural and consonant harmony possible results. Because of this the fundamental has been called the *key* of the scale. When chords are built of the other intervals, the impression of various degrees of mistuning results. The just scale is appropriate only for music of simple chord structure, like that of the classical or earlier periods of music, or of much popular music.

To enable an instrument of fixed pitch, such as the harpsichord, to play chords from all keys equally well, the *tempered scale* was devised. It divides the octave into 12 evenly spaced pitches. Thus the frequency ratio of any two adjoining semitones equals $2^{1/2}$, or approximately 1.0594. The tempered scale allows free mixing of chords and keys with uniform and only minor mistuning.

The tempered scale has been the dominant scale in Western music for nearly three centuries, and is so familiar that its mistuning is usually unnoticed. Its reduction of the just scale's 20 pitches to 12 is accomplished by collapsing the eight closest pitch pairs (usually adjoining sharps and flats) into eight single pitches.

The current frequency standard for music fixes the A pitch in the fourth piano octave to 440 Hz. The other A pitches differ by multiples or factors of 2 from A440. The frequencies of all A pitches in the piano keyboard range are listed in Table 7.3.

The frequency range of the human ear is much wider, from about 15 to 20,000 Hz for an unimpaired young person. This range narrows with age. SID produces all

TABLE 7.3 FREQUENCY OF A PITCHES

Pitch	Frequency
A0	27.5
A1	55
A2	110
A3	220
A4	440
A5	880
A6	1760
A7	3520

fundamental frequencies from 0 to approximately 4 kHz, with harmonics present to well above 12 kHz.

Table 7.4 gives the frequency ratios between the key pitch and each scale tone within one octave for both the just and the tempered scales. Examples of each interval are given in the key of A.

Specific frequencies for pitches in the tempered scale can be calculated by applying tempered ratios to the appropriate A pitch in Table 7.4. For example, E♭ above A3 has the frequency $1.4142 \times 220 = 311.124$ Hz in the tempered system. Specific frequencies for pitches in the just scale are most easily calculated by obtaining the frequency of the key pitch from the tempered scale and then applying the appropriate just-scale ratio to it.

Other scales have been used in this century. Departures from the tempered scale include divisions of the smallest tempered interval into halves, thirds, and sixths. These correspond to pitch ratios of $2^{(1/24)}$, $2^{(1/36)}$, and $2^{(1/72)}$, respectively. Such scales, called *microtonal,* allow for delicate and piquant harmonies. Alternatively, scales can be built from the unrestricted use of rational fraction intervals, that is, from intervals whose pitch ratios are related to the key by any fractions having integer numerators and denominators. The latter system owes its heritage to the harmonic-based just scale.

TABLE 7.4 FREQUENCY RATIOS WITHIN AN OCTAVE FOR THE JUST AND TEMPERED SCALES

Interval	Just ratio	Tempered ratio
Unison (A–A)	1/1 = 1.0000	$2^{(0/12)}$ = 1.0000
Augmented unison (A–A♯)	25/24 = 1.0417	$2^{(1/12)}$ = 1.0595
Minor second (A–B♭)	16/15 = 1.0667	= 1.0595
Major second (A–B)	9/8 = 1.1250	$2^{(2/12)}$ = 1.1225
Augmented second (A–B♯)	75/64 = 1.1719	$2^{(3/12)}$ = 1.1892
Minor third (A–C)	6/5 = 1.2000	= 1.1892
Major third (A–C♯)	5/4 = 1.2500	$2^{(4/12)}$ = 1.2599
Diminished fourth (A–D♭)	32/25 = 1.2800	= 1.2599
Augmented third (A–C♯♯)	125/96 = 1.3021	$2^{(5/12)}$ = 1.3348
Perfect fourth (A–D)	4/3 = 1.3333	= 1.3348
Augmented fourth (A–D♯)	45/32 = 1.4062	$2^{(6/12)}$ = 1.4142
Diminished fifth (A–E♭)	36/25 = 1.4400	= 1.4142
Perfect fifth (A–E)	3/2 = 1.5000	$2^{(7/12)}$ = 1.4983
Augmented fifth (A–E♯)	25/16 = 1.5625	$2^{(8/12)}$ = 1.5974
Minor sixth (A–F)	8/5 = 1.6000	= 1.5974
Major sixth (A–F♯)	5/3 = 1.6667	$2^{(9/12)}$ = 1.6820
Augmented sixth (A–F♯♯)	225/128 = 1.7578	$2^{(10/12)}$ = 1.7820
Minor seventh (A–G)	9/5 = 1.8000	= 1.7820
Major seventh (A–G♯)	15/8 = 1.8750	$2^{(11/12)}$ = 1.8878
Diminished octave (A–A♭)	48/25 = 1.9200	= 1.8878
Augmented seventh (A–G♯♯)	125/64 = 1.9531	$2^{(12/12)}$ = 2.0000
Octave (A–A 8va)	2/1 = 2.0000	= 2.0000

The waveform attribute can now be described in terms of sound intensity and harmonic composition.

Waveform

For any cyclical sound the graph or function of the pressure variation with time is called the *waveform*. Waveforms can take on an infinite variety of patterns other than the sine curve.

In the preceding section we stated that every possible sound consists of some combination of sine waves having individual intensities. It can further be shown that every cyclical sound consists of a combination of a fundamental sine pitch and some subset of its harmonics, with all frequencies having individual intensities.

The frequency of a waveform is the same as that of its fundamental. Combining sine waves of different frequencies means adding together their individual pressure offsets from the ambient pressure. At times these offsets will cancel, leaving ambient pressure at the measuring point. At other times they will reinforce each other for a larger pressure offset than would result from any individual sine component.

A few waveforms are especially prevalent. These include the square wave, the triangle wave, and the ramp wave.

Graphing the *square wave* produces the result shown in Fig. 7.5. The square wave contains the fundamental pitch and the *odd harmonics*. An odd harmonic is one whose frequency is an odd multiple of the fundamental. In terms of string harmonics, the square wave contains just those frequencies for which the number of nodes m is odd. Further, the intensity of each harmonic equals $1/m$. Thus the full square-wave definition is

$$P = 1/1 \times \sin(2\pi f) + 1/3 \times \sin(2\pi \cdot 3f) + 1/5 \times \sin(2\pi \cdot 5f) + \cdots$$

Another waveform, the *triangle wave,* also contains only the fundamental pitch and the odd harmonics. However, the relative intensity of the harmonics and

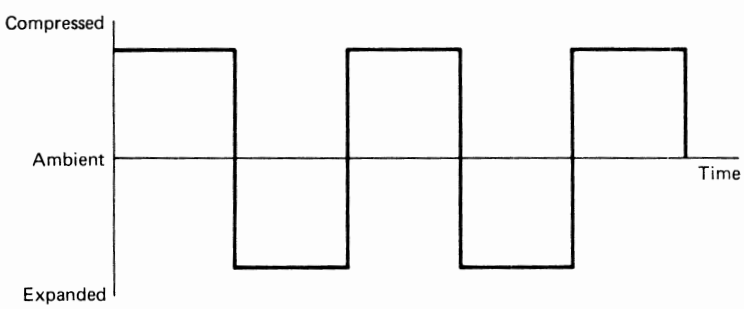


Figure 7.5

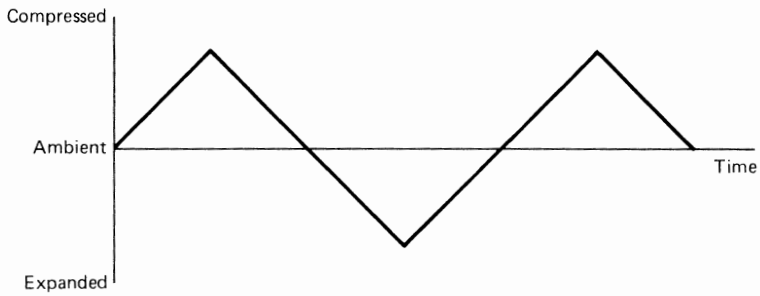


Figure 7.6

even their phase relationships are different than in the square wave (phase will be discussed in the next section). The triangle wave is shown in Fig. 7.6.

For practical purposes, the most useful thing to remember about the triangle wave is that the intensities of its harmonics fall off much more rapidly than do those of the square wave. Instead of a $1/m$ degradation, intensity decreases proportionately to $1/(m^2)$. This makes the fundamental pitch more dominant over the harmonics in the triangle wave.

The *ramp wave,* alone of these three waveforms, contains the fundamental and all harmonics (Fig. 7.7). The ramp wave contains the same harmonics in the same intensities as the square wave, adding to them the even harmonics, also with $1/m$ intensities. Thus the ramp-wave definition is

$$P = 1/1 \times \sin(2\pi f) + 1/2 \times \sin(2\pi \cdot 2f) + 1/3 \times \sin(2\pi \cdot 3f) + \cdots$$

Any of these waveforms can be represented by a *spectrum plot,* a graph of the harmonics and harmonic intensities present in a sound. The spectrum plot for a 1000-Hz ramp wave, with harmonic intensity scaled from 0 to 1.0, is shown in Fig. 7.8. Spectrum plots for the other two waveforms can be made from the information on their harmonic frequencies and intensities already given in this section.

Interestingly, the ear does not hear a waveform as a single sound; instead, it

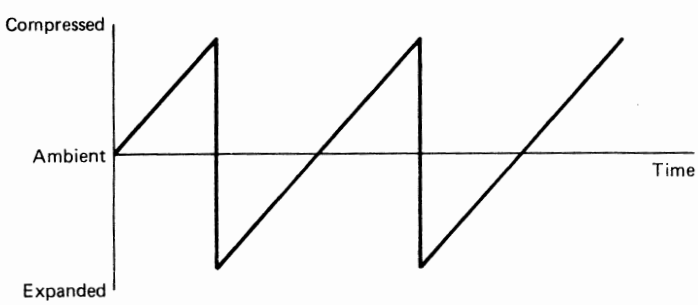


Figure 7.7

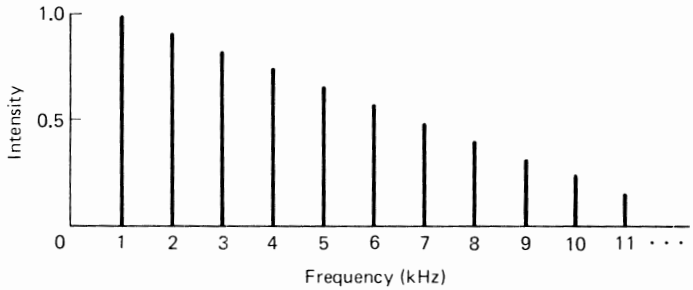


Figure 7.8

analyzes the waveform and recreates the harmonic content of the wave. With training, listeners can identify harmonics in the waveform as individual pitches!

Nevertheless, the combined effect of the harmonics in their relative intensities is to color the fundamental. Musicians call this the tone quality, tone color, or timbre of the fundamental pitch. Removing a single harmonic from a waveform usually will not change the timbre of the sound appreciably, because timbre is a function of the relationships between all the harmonics. The overall shape of these relationships is hardly affected by changes to any one harmonic.

Waveform is the attribute that distinguishes the sound of one instrument from another when both are heard sustaining long tones. It can also distinguish the sustained tone of a new and artificial instrument of your creation from that of all natural instruments.

Phase

Phase is the measure of the time separation between the same point in the cycles of a subject wave and a simultaneous reference wave. A wave cycle can be divided into 360 degrees or 2π radians, like a circle, and the difference between the same point in the two waves is usually expressed in these units. For instance, if the maximum point of the subject wave falls 1/4 cycle after the reference wave maximum, the subject wave is said to trail the reference wave by 90 degrees. This is shown in Fig. 7.9 for two sine waves. The subject wave is dashed and the reference wave is solid.

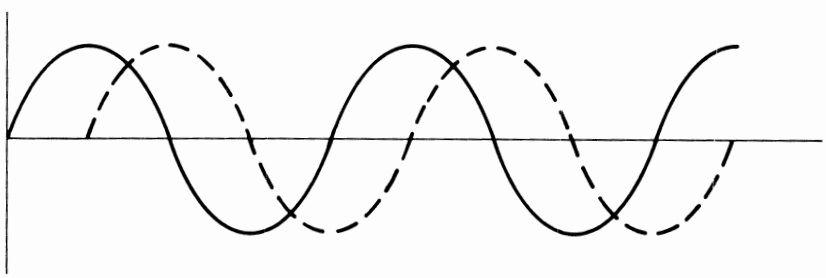


Figure 7.9

As mentioned in the preceding section, the ear breaks sounds into harmonic frequencies with relative intensities. Its emphasis on individual frequencies and intensities leaves it nearly insensitive to the relative phases of the harmonics. Thus the phase attribute can usually be ignored in sound synthesis.

WAVE MODULATION

To use sound to communicate requires changing the values of one or more of the four sound wave attributes with time. The changing of a sound attribute is called *modulation.* SID's sound-molding abilities, and the detectable sound qualities of natural instruments as well, can be explained as modulations of the first three wave attributes: intensity, frequency, and waveform. In this section we discuss the registers and capabilities available with SID for modulating sounds. The last major section of this chapter, on sound programming, explains how these capabilities are combined and used from programs to generate the sounds you want.

With natural instruments modulation is under the control of a player. Since human beings are imperfect, a player's intentional modulations of intensity, frequency, and to a lesser extent waveform, are departed from slightly at random intervals. This effect is due to entropy, the natural law of disorder discussed in Chapter 2. The most realistic simulation of natural sounds will also subtly modify the smooth sweep of modulation at random, relatively frequent intervals.

Sound is usually modulated to obtain a subjective effect. Therefore, this section will separate SID's features by the subjective form of the wave attribute they modify. In the order we discuss them, these subjective attributes are loudness, pitch, and timbre.

Loudness

SID provides three types of loudness control: overall, voice, and harmonic. In *overall loudness control,* a program writes a nibble into bits d0 through d3 of SID location D418 to change the output volume of all three voices simultaneously. These four bits allow for 16 settings, from 0 to 15. 0 selects silence, 15 selects full volume, and the 16 values smoothly divide the 48-dB range into approximately 3-dB increments (again, the actual loudness range and division depend on the TV or stereo amplification applied to SID's signal). From the equation relating intensity to loudness, each 3-dB increase corresponds to a doubling in intensity.

In *voice loudness control,* the volume of a voice is changed over time, within the limits set by the overall volume control, using one of three *envelope generators.* Each voice has been permanently assigned its own envelope generator.

The envelope generators mold a sound in four stages, which are based on the characteristics of natural sounds. The volume of a simplified natural sound changes through four stages: attack, decay, sustain, release. Together, these stages are sometimes called the *ADSR cycle.* In the *attack stage,* the application of energy to a

physical object causes it to begin vibrating and emitting sound. Typically, this initial energy pushes the sound intensity to a maximum level in a few milliseconds.

Then as the effect of the initial energy application fades and the object settles into a vibrating pattern, the sound intensity falls to an intermediate level. This rapid falling is called the *decay stage,* and it also typically lasts for a few milliseconds.

The *sustain stage* is the time in which the object is vibrating in its normal manner, with little falling off in volume. Finally, in the *release stage* the object's inherent resistance to vibration overcomes the decreasing stored energy of vibration, and the object's movement and sound die out. The four stages are graphed as an envelope in Fig. 7.10.

If a program were given complete control over voice volume, simulating this envelope would require a lot of code, including timing loops for each volume level in each stage. By sacrificing an entirely general volume capability, SID has provided the most natural volume pattern possible with minimal program and CPU work.

SID allows much flexibility in how long each stage of the envelope is played. Times approaching 1 second for the first two phases are possible, and although distinctly unnatural, they provide unusual creative possibilities. For instance, a sound that rises slowly and ends abruptly resembles the backward replay of a tape recording of a natural instrument.

Seven SID registers have been dedicated to each voice for controlling the voice's independent sound characteristics. Voice 1 registers extend from D400 to D406, with voice 2 from D407 to D40D, and voice 3 from D40E to D414. The envelope generator for each voice is accessed through three of these registers.

The attack-, decay-, and release-stage lengths for each voice are selected through three nibbles in two ADSR registers. The sustain-stage length is controlled through a bit in another register, as we shall see shortly. The sustain volume is selected through the remaining nibble in the ADSR registers, as a fraction of the attack phase maximum (from 0 for silence to F for maximum). The maximum volume during the attack phase is the same as the maximum overall volume.

The ADSR register locations for each voice are shown in Table 7.5. The effects of each nibble value on the ADSR envelope are shown in Table 7.6.

Once the ADSR and other parameters for a sound have been loaded into SID, the sound is triggered by setting a gate bit in another voice register to 1. The attack

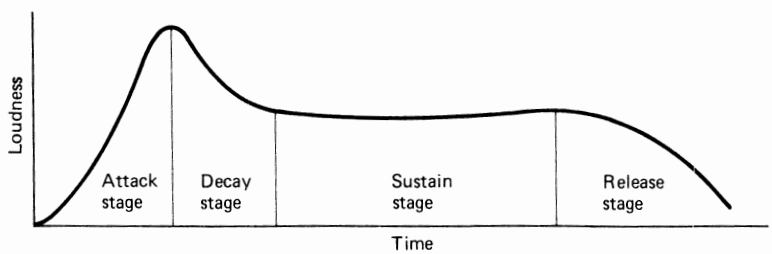


Figure 7.10

TABLE 7.5 ADSR PHASE REGISTERS

Phase	Voice	Address	Bits
Attack	1	D405	d4–d7
	2	D40C	
	3	D413	
Decay	1	D405	d0–d3
	2	D40C	
	3	D413	
Sustain	1	D406	d4–d7
	2	D40D	
	3	D414	
Release	1	D406	d0–d3
	2	D40D	
	3	D414	

phase begins immediately, progressing through the delay and sustain phases. Volume remains at the sustain level until the gate bit is reset to 0, which triggers the release phase. However, either triggering action can be taken at any point in the ADSR cycle, allowing the sound to be restarted or cut short before the normal time. The gate bits for voices 1, 2, and 3 are in bit d0 of locations D404, D40B, and D412, respectively.

Voice 3 also has its own on/off bit, bit d7 of location D418 (0 = On), which allows the voice to generate its programmed sound without sending it out for audio amplification. This is the crudest level of voice control available. The reasons for its existence will be discussed shortly.

TABLE 7.6 ADSR CONTROL VALUES

Nibble	Attack time	Decay/release time
0	2 ms	6 ms
1	8 ms	24 ms
2	16 ms	48 ms
3	24 ms	72 ms
4	38 ms	114 ms
5	56 ms	168 ms
6	68 ms	204 ms
7	80 ms	0.24 s
8	0.10 s	0.30 s
9	0.25 s	0.75 s
A	0.50 s	1.5 s
B	0.80 s	2.4 s
C	1 s	3 s
D	3 s	9 s
E	5 s	15 s
F	8 s	24 s

In *harmonic loudness control,* the relative loudness of the harmonics is varied in a limited but effective way. A program selects a pitch as a reference frequency, and the volume of all pitches above, below, or both above and below of the reference pitch are suppressed in proportion to their separation from the reference. The farther a given pitch is from the reference pitch, the more it will be suppressed. This technique is called *filtering.* It is mentioned in this section because in natural instruments each harmonic has a separate volume (e.g., ADSR) envelope, which can be simulated on SID by varying filtering as the ADSR envelope changes. Although most applications do not require this level of realism or effort, consider experimenting with changing the filter reference frequency as a sound is played. This has the effect of modulating a sound's timbre, so we will discuss the filter registers and the values that can be written into them in the latter part of the timbre section.

Pitch

SID produces pitches from 0 to 3996 Hz. Another two of each voice's seven registers are used to select the voice's pitch. The 16 pitch bits divide the 3996 Hz into 2^{16} or 65,536 steps of 0.06097 Hz each. In practice one usually starts with a frequency and calculates the appropriate register values. SID requires these values in the form of an integer quotient and remainder:

high register = DIV[ROUND(frequency/0.06097)/256]
low register = MOD[ROUND(frequency/0.06097)/256]

Thus, the integer quotient goes into the high-register and the integer remainder goes into the low register. The division frequency/0.06097 is rounded so that the MOD operation will produce an integer result. To calculate the register values for the pitch A at 440 Hz the equations are

ROUND(440/0.06097) = 7217

high register = DIV[(7217)/256] = 28
low register = MOD[(7217)/256] = 49

The locations for the high and low registers for all three voices are listed in Table 7.7.

The most obvious type of pitch modulation is to change between scale pitches to form a melody. The programmer can form a table of the frequency-register values for all the pitches in the scale, and then store the melody as a series of table-index values.

Another type of pitch modulation varies the frequency of a single sound. The most common musical examples of this are the vibrato and the portamento. The *vibrato* is a wavering of pitch around a center frequency. Instrumentalists commonly use cycles lasting about 1/7 second each, with frequency reaching 1/4 scale step above and below the center frequency. Vibrato adds an aura of warmth and realism to a computer-generated sound. To use it from a program, the pitch must be

TABLE 7.7 VOICE LOCATIONS

	Voice 1	Voice 2	Voice 3
High register	D401	D408	D40F
Low register	D400	D407	D40E

changed by small amounts within the 1/4-step limits at regular time intervals. As we shall see, the time interval can be generated by letting the sound-handling routine be interrupt driven (i.e., called by the system clock or raster interrupt).

The *portamento* is a sliding of frequency between two terminal pitches. Trombones and violins are two types of instruments that can perform true portamenti. With its closely spaced discrete frequencies, SID can perform a convincing simulation of a true portamento.

These two techniques, and any other variation on this type of pitch modulation, require two actions of a program. First, the program must keep a record of either the original or the current frequency-register values for a sound. This is necessary because almost all SID registers are write only; voice frequency cannot be read from SID. Second, the program must generate or obtain an offset for changing the original or current frequency into the next frequency.

The program produces a new frequency from the above two data items according to one of the following relationships:

new frequency = current frequency + incremental offset
new frequency = original frequency + new total offset

Either form can generate a vibrato or portamento. Depending on the source of the offset value, one or the other form will be easiest and fastest to use. If the program generates the offset according to a mathematical function, like simple incrementing and decrementing, the first form will often be best. This method is well suited, for example, for wide portamenti. Alternatively, the program can obtain an offset value from either of two SID registers. This type of offset fits naturally into the second relationship. The first type of offset needs no further explanation, so we will go on to the second.

One source of an offset value is SID's waveform information. SID produces four waveforms or timbres: pulse, triangle, ramp, and noise. Modulation of these waveforms is the topic of the next section, but their use in modulating frequency can be discussed here.

SID represents each waveform with a changing eight-bit value. The pulse wave consists of alternating 00 and FF values. The triangle wave consists of values that increment to FF and decrement back to 00 repeatedly. The ramp wave consists of values that increment to FF and wrap around to 00 to increment again (i.e., modulo FF incrementing). Noise consists of values changing randomly. The values of all four waveforms change at the rate of their frequency (e.g., a 1-kHz waveform changes waveform values 1000 times a second).

The waveform value for voice 3 is made available through a register called oscillator 3, at location D41B. This value can be read, scaled (by shifting right or left), and converted to a 16-bit number (by appending a MSByte of 00) for adding to the sound's original frequency. Note that the original frequency must be lower by half the offset range to center the variations around a desired pitch. For a vibrato, voice 3 should be set to a triangle waveform of approximately 7 Hz frequency, and shifted right to scale it for a 1/4-step offset in the frequency range of the desired pitch. Note that voice 3 can be set to the noise waveform to obtain random numbers from oscillator 3 for nonmusical applications.

For a waveform to modulate a voice's frequency audibly, it must be of a very low frequency. This makes the waveform source, voice 3, relatively useless as an audible voice. Thus, in this application the voice 3 audible output is usually turned off by setting the d7 bit of location D418 to 1, as described earlier.

A second source for a frequency offset is the voice 3 ADSR envelope. SID makes its values—rising from 00 to FF on being gated, falling to an intermediate value, and falling to 00 on the second gating—available through a register called envelope 3, at location D41C. Like oscillator 3, this register must be read, scaled, converted to 16 bits, and added to the frequency-register's value for the modulated voice. Since the audible output of voice 3 is seldom aesthetic in this role, voice 3 output is also usually turned off with bit d7 of D418.

Timbre

The four waveforms generated by SID produce four classes of timbre. One of each seven voice registers contains timbre selection and other control bits. These control registers are located at D404, D40B, and D412 for voices 1, 2, and 3, respectively.

The contents of each control register are outlined in Table 7.8. The unfamiliar bits will be explained shortly. Each bit's function is selected with a 1 value.

Bits d4 through d7 select the voice waveform. If the pulse waveform is selected, a 12-bit pulse-width value must be placed in a pulse-width register pair belonging to the voice. A pulse-width value of 800 hex divides each wave period

TABLE 7.8 CONTROL REGISTER

Bit	Purpose
0	ADSR gate bit
1	Sync with prior voice
2	Ring mod with prior voice
3	Voice disable
4	Triangle waveform select
5	Ramp waveform select
6	Pulse waveform select
7	Noise waveform select

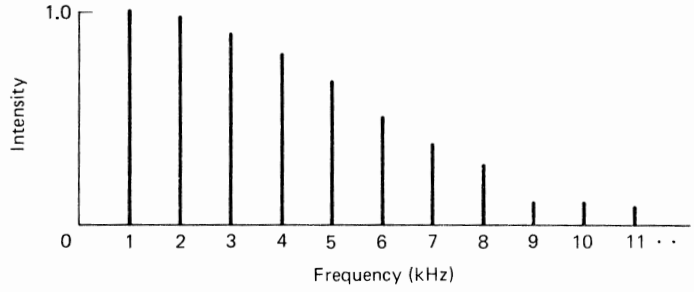


Figure 7.11

evenly between high and low wave output values (the parameter discussed under pitch modulation). An evenly divided pulse wave is also called a *square wave.* With a pulse width of 000 the wave output value remains at 00 all the time, and with a pulse width of FFF the wave output remains at FF constantly. Other values divide the pulse cycle unevenly, for a more complex timbre than that of the square wave. The general expression for the percentage of cycle time that the wave output value is high is

% of cycle that waveform output value is high = 100 × [(pulse width value)/4096]

A pulse wave with a 10% pulse width has all harmonics strongly present. The spectrum for a pulse of 10% width at 1 kHz is shown in Fig. 7.11. The pulse-width register locations for the three voices are listed in Table 7.9.

If more than one waveform is selected in the control register, the different waveform values are logically ANDed to produce a new waveform. Selecting the pulse wave and any of the other waveforms has an interesting effect, for instance.

A 1 value in the control register voice-disable bit (d3) locks the waveform oscillator at a 00 value until the bit is reset. Gate bit d0 of the control register is familiar from our discussion of the ADSR envelope. The remaining two control register bits, bits d2 and d1, select ring modulation and voice synchronization, respectively. These functions combine the given voice with another voice for special effects. Commodore is tight-lipped about the actual effects of these functions, so you will have to experiment with them to become familiar with the sounds you can produce. Selecting either function for voice 1 combines voice 1 with voice 3. Selecting either function for voice 2 combines voice 2 with voice 1. Finally, selecting either

TABLE 7.9 PULSE-WIDTH REGISTERS

Position	Contents	Address
Low byte	Lower byte of PW value	D402(V1),D409(V2),D410(V3)
High byte:		
d0–d3	Upper nibble PW value	D403(V1),D40A(V2),D411(V3)
d4–d7	Unused	

function for voice 3 combines voice 3 with voice 2. Try different combinations of frequency and waveform in each of the voices to explore the many interesting effects that are possible.

However the waveform of a voice has been generated, it can then be further modulated by filtering its harmonic structure. As stated in the preceding section, filtering drops the volume of frequencies on one or both sides of a reference or cutoff frequency.

Four types of filtering are possible with SID. *Lowpass* filtering suppresses all frequencies above a cutoff frequency. *Highpass* filtering suppresses all frequencies below a cutoff frequency. *Bandpass* filtering suppresses all frequencies on either side of the cutoff. *Notch* filtering is a combination of lowpass and highpass filtering to suppress the cutoff and immediately surrounding frequencies. The effects of these filterings are shown in Fig. 7.12, with "gain" signifying the change in loudness level.

Frequencies above a lowpass cutoff or below a highpass cutoff are suppressed by 12 dB for each octave separating them from the cutoff. Frequencies around a bandpass cutoff are suppressed by 6 dB per octave of separation. The effect shown for the notch filter is only approximate.

The filtering type is selected by one or more of three bits sharing the overall output-volume register, at D418. Setting bit d4 to 1 selects lowpass filtering, bit d5 similarly selects bandpass, bit d6 selects highpass, and setting both bit d4 and d6 selects notch filtering.

The cutoff frequency is selected by placing an 11-bit value in locations D415 and D416. Commodore makes no claims as to the preciseness with which a frequency can be selected; it claims only that a range of cutoff frequencies from approximately 30 to 10,000 Hz is linearly divided 2^{11} ways. Taken literally, this divides the range into 4.868-Hz intervals. Assuming that this value is accurate, an approximate cutoff register value can be calculated from the desired cutoff frequency as follows:

$$\text{cutoff register value} = \text{ROUND}[(\text{frequency} - 30)/4.868]$$

The lower 3 bits of the cutoff value go into d0 through d2 of location D415. The upper byte of the cutoff value goes into D416.

Frequencies in the immediate neighborhood of the cutoff frequency can be boosted above their normal loudness by altering what Commodore calls the filter's "resonance" (we point to Commodore's usage because the word "resonance" normally refers to the sharpness of volume suppression of all affected frequencies around the cutoff frequency, a meaning that does not apply to SID). A nibble value from 0, for no boost, to F, for maximum boost, can be placed into bits d4 through d7 of location D417. For the clearest sound this value should normally be set to F.

Just as the contents of oscillator 3 and envelope 3 were retrieved and used to modulate a voice's frequency, they can be used to modulate the filter's cutoff frequency. The resulting timbre changes are dramatic, but difficult to describe with words. Anyone interested in unusual tone-color effects should experiment with

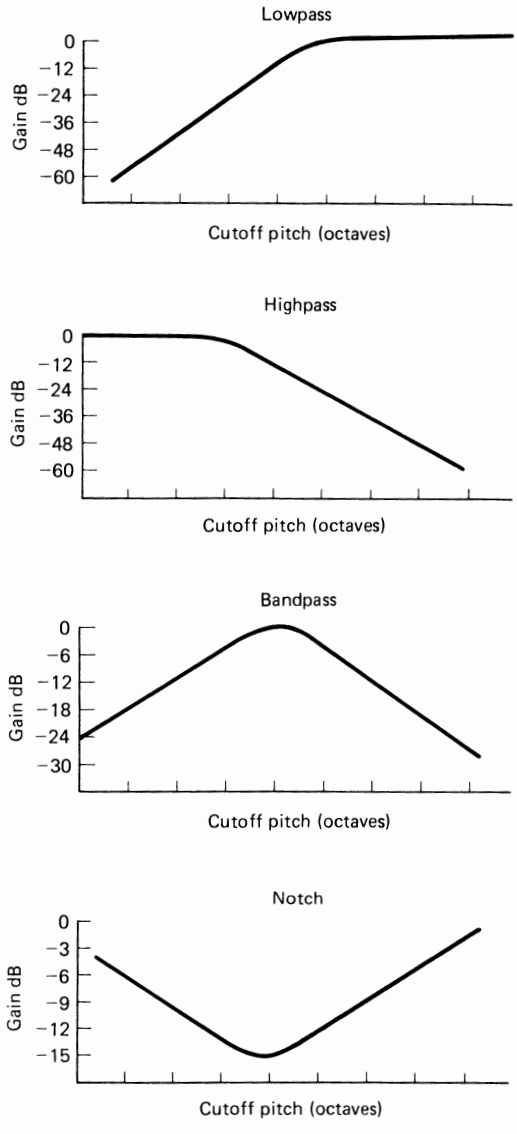


Figure 7.12

modulating the cutoff frequency with many different waveforms and ADSR envelopes.

SID has only one filter, but each voice can be routed either through or around it. The filter bypass bits d0 through d2 of location D417 represent voices 1 through 3, respectively. A 1 bit value routes the voice around the filter, and a 0 value routes it through to be filtered.

SOUND PROGRAMMING

The key to achieving the sounds you want with SID is to experiment with modulation. Try combining the different types of sound modulations in different ways and see which combinations produce the most effective results. For instance, you could combine a triangle waveform with a vibrato pitch change and a slow attack but fast release.

We have discussed how natural instruments can be simulated by varying volume, pitch, and timbre. Unnatural sounds can be produced the same way. Voices with different wave characteristics can be played simultaneously to enrich the resulting timbre.

By being familiar with the underlying wave nature of the sound that you are handling, you will be able to vary different wave attributes in an intelligent way. As you become experienced in SID's use, you will be able to start with sounds that are closer to your intentions, to attain desired sounds more quickly than would otherwise be possible, and perhaps most important, be able to explore sounds you cannot mentally imagine, which may have the greatest effects of all.

The preferred method for handling sound from a program resembles the preferred method for handling screen graphics; the program prepares for I/O operations and then utilizes an IRQ interrupt handler. To review how the IRQ handler works, see the animation section of Chapter 6. The IRQ can be driven by the raster, which is convenient if both sound and screen changes are being performed, or by the system-cycle IRQ from CIA chip #1 (to review the system cycle, see the kernel clock section in Chapter 5). With a raster IRQ and video processing, the video portion of the work should be performed first because it is the most time critical. Whichever IRQ is used to drive the audio interrupt handler, for timing preciseness it should usually be the only IRQ enabled.

The program's duties in preparing for audio I/O are to write the value 0 to all SID registers (locations D400 through D41C hex), and then to turn the overall volume (location D418, low nibble) full on (value 0F hex). This places SID in a startup condition, ready to be configured to produce sound.

The interrupt handler's duties include everything associated with producing the sounds. This consists of setting up the filter type, cutoff, and resonance; routing the voice or voices through the filter; setting up and altering the voice frequency, pulse-width and ADSR values for the current state of the sound; checking for cues to begin a sound (e.g., a keyboard input or a specific timer value); setting up the voice control register with the selected waveform bit and with d0 = 1 to start the attack stage; timing the ADS stages; resetting the d0 bit of the voice-control register to start the release stage; and turning the voice off by writing a 0 into the voice-control register when the sound is done. Obviously, not all these activities will be done in any one activation of the interrupt handler. Using a 1/60-second raster or CIA IRQ to drive the interrupt handler and a counter in the handler code will allow determining when each activity should be performed.

The pseudocode for the duties of an audio interrupt handler is given below.

Not all pseudocode lines will be performed for every sound, but together they describe the general case. The handler duties cover the lifetime of a sound, which is usually far longer than 1/60 second, which is the time between two raster or system IRQs. Hence these duties must be divided and the parts performed over many interrupt handler executions. The timing counter we just mentioned can be incremented each time the interrupt handler executes, and the various actions in the handler pseudocode can be associated with different counter values. The assembly code for an audio handler resembles a CASE construct, with the CASE processing options being the actions described in the following pseudocode lines or in lines produced by expanding the pseudocode. Over time and many execution cycles, the interrupt handler will work its way through the actions described in the pseudocode below.

```
Produce a sound
  Set up the filter type, cutoff, and resonance (D415-D418)
  Route the voices through the filter (D417)
  Set up the voice frequencies, pulse widths, and ADSR envelope
  Check for cue to begin sound
  IF time to begin sound
    THEN Select voice waveform and start its attack phase
         LOOP
              Modulate the voice frequency, loudness, or timbre
              Modulate overall loudness
              IF Sustain-phase time has just completed
                 THEN Start the release phase
              ENDIF
           EXITIF Sound has died out
         ENDLOOP
    ENDIF
END Produce a sound
```

We conclude by providing in Table 7.10 some example SID values for simulating common musical instruments. Filter settings for the filtered instruments

TABLE 7.10 INSTRUMENT SETTINGS

Instrument	A	D	S	R	Waveform	Filter	Cutoff
Piccolo	5	3	A	4	Triangle		
Flute	5	5	5	3	Triangle	Lowpass	50
Clarinet	5	4	4	2	Pls(800)	Lowpass	50
Oboe	3	3	A	4	Pls(800)	Lowpass	50
Bassoon	3	5	9	5	Triangle	Lowpass	40
Trumpet	3	6	3	5	Sawtooth	Lowpass	50
Cornet	5	3	6	4	Sawtooth	Lowpass	50
French horn	5	3	6	4	Pls(500)	Lowpass	40
Bass drum	0	B	0	0	Triangle		

are for notes between middle C, or C5, and C7. Resonance equals F hex for all filtered cases. The pulse-width value is shown for pulse waveform cases. All values are hexadecimal.

These values omit vibrato, timbre change, and other dynamic modulations, yet still yield recognizable sounds. With SID's flexibility, most familiar sounds can be recognizably simulated given the right register values and enough dynamic control of timbre and frequency by the program.

Exercise:

Answer the following questions to review C64 sound synthesis.

(a) What are the four major attributes of sound waves?

(b) Explain the modulation facilities of the C64.

FOR FURTHER STUDY

Subject	Book	Publisher
Music on the C64	*Music and Sound for the Commodore 64* by Bill L. Behrendt	Prentice-Hall, Inc.

A

ANSWERS TO SELECTED EXERCISES IN CHAPTER 1

(b) 144 (c) 47 (d) 215 (e) 1111 1101 (f) 0011 0100
(g) 1001 0011 (h) 1111 1110 (i) 1 1010 1010
(j) 1 1000 1011 (k) 1110 (l) 1101 0100 (m) 0001 0011
(n) 100101.11 (o) 11.8125 (p) 10110110.0011
(q) 1101.01 (s) .11001001 (t) .11010101 (u) 10111111
(v) 1010 (w) 11011010 (x) AE hex (y) 1011 1101 0111 1111
(z) 5CBE hex (aa) 1 0101 0100

B

THE SCREEN DISPLAY CODES

Placing values from one of the screen display codes directly into screen-memory locations will cause the corresponding characters to be displayed as long as the SID chip is in character mode (see Chapter 6). Code 1 is selected by sending the value 0E hex out the screen channel using kernel module CHROUT. Code 2 is selected by sending value 8E hex out the screen channel.

In all these coded values the d7 bit equals 0. Setting the d7 bit reverses the two bit colors in the character. The predefined characters and their coded values are given in the following table. Use code 1 character where code 2 is blank.

Character		Value	
Code 1	Code 2	h	(d)
@		0	(0)
A	a	1	(1)
B	b	2	(2)
C	c	3	(3)
D	d	4	(4)
E	e	5	(5)
F	f	6	(6)
G	g	7	(7)
H	h	8	(8)
I	i	9	(9)
J	j	A	(10)
K	k	B	(11)
L	l	C	(12)
M	m	D	(13)
N	n	E	(14)

Character		Value	
Code 1	Code 2	h	(d)
O	o	F	(15)
P	p	10	(16)
Q	q	11	(17)
R	r	12	(18)
S	s	13	(19)
T	t	14	(20)
U	u	15	(21)
V	v	16	(22)
W	w	17	(23)
X	x	18	(24)
Y	y	19	(25)
Z	z	1A	(26)
[		1B	(27)
£		1C	(28)
]		1D	(29)

Character		Value	
Code 1	Code 2	h	(d)
↑		1E	(30)
←		1F	(31)
		20	(32)
!		21	(33)
"		22	(34)
#		23	(35)
$		24	(36)
%		25	(37)
&		26	(38)
'		27	(39)
(		28	(40)
)		29	(41)
*		2A	(42)
+		2B	(43)
,		2C	(44)
-		2D	(45)
.		2E	(46)
/		2F	(47)
0		30	(48)
1		31	(49)
2		32	(50)
3		33	(51)
4		34	(52)
5		35	(53)
6		36	(54)
7		37	(55)
8		38	(56)
9		39	(57)
:		3A	(58)
;		3B	(59)
<		3C	(60)
=		3D	(61)
>		3E	(62)
?		3F	(63)
		40	(64)
♠	A	41	(65)
	B	42	(66)
	C	43	(67)
	D	44	(68)
	E	45	(69)
	F	46	(70)
	G	47	(71)
	H	48	(72)
	I	49	(73)
	J	4A	(74)
	K	4B	(75)
	L	4C	(76)
	M	4D	(77)
	N	4E	(78)

Character		Value	
Code 1	Code 2	h	(d)
	O	4F	(79)
	P	50	(80)
●	Q	51	(81)
	R	52	(82)
♥	S	53	(83)
	T	54	(84)
	U	55	(85)
	V	56	(86)
	W	57	(87)
♣	X	58	(88)
	Y	59	(89)
♦	Z	5A	(90)
		5B	(91)
		5C	(92)
		5D	(93)
		5E	(94)
		5F	(95)
		60	(96)
		61	(97)
		62	(98)
		63	(99)
		64	(100)
		65	(101)
		66	(102)
		67	(103)
		68	(104)
		69	(105)
		6A	(106)
		6B	(107)
		6C	(108)
		6D	(109)
		6E	(110)
		6F	(111)
		70	(112)
		71	(113)
		72	(114)
		73	(115)
		74	(116)
		75	(117)
		76	(118)
		77	(119)
		78	(120)
		79	(121)
	√	7A	(122)
		7B	(123)
		7C	(124)
		7D	(125)
		7E	(126)
		7F	(127)

C

ASCII AND XASCII CODES

In the following table, ASCII codes are listed under the "std." column, and special XASCII codes are listed under the "64" column. The "standard meaning" column applies to the ASCII code where ASCII and XASCII differ.

Character		Value			Character		Value		
Std.	64	hex	(dec)	Standard meaning	Std.	64	hex	(dec)	Standard meaning
ENQ		5	(5)	Enquiry	DC3		13	(19)	transfers with
BEL		7	(7)	Ring bell	DC4		14	(20)	printers,
BS		8	(8)	Backspace					etc.
					ESC		1B	(27)	Escape
HT		9	(9)	Horiz. tab			1C	(28)	
LF		A	(10)	Linefeed			1D	(29)	
VT		B	(11)	Vert. tab					
FF		C	(12)	Form feed			1E	(30)	
CR		D	(13)	Carriage Return			1F	(31)	
		E	(14)		(Both codes)				
					SP		20	(32)	
DC1		11	(17)	Device	!		21	(33)	
				controls,	"		22	(34)	
DC2		12	(18)	for coor-	#		23	(35)	
				dinating	$		24	(36)	

Character		Value		Standard meaning
Std.	64	hex	(dec)	
%		25	(37)	
&		26	(38)	
'		27	(39)	Apostrophe
(		28	(40)	
)		29	(41)	
*		2A	(42)	
+		2B	(43)	
,		2C	(44)	Comma
-		2D	(45)	Hyphen
.		2E	(46)	
/		2F	(47)	
0		30	(48)	
1		31	(49)	
2		32	(50)	
3		33	(51)	
4		34	(52)	
5		35	(53)	
6		36	(54)	
7		37	(55)	
8		38	(56)	
9		39	(57)	
:		3A	(58)	
;		3B	(59)	
<		3C	(60)	
=		3D	(61)	
>		3E	(62)	
?		3F	(63)	
@		40	(64)	
A		41	(65)	
B		42	(66)	
C		43	(67)	
D		44	(68)	
E		45	(69)	
F		46	(70)	
G		47	(71)	
H		48	(72)	
I		49	(73)	
J		4A	(74)	
K		4B	(75)	
L		4C	(76)	
M		4D	(77)	
N		4E	(78)	
O		4F	(79)	
P		50	(80)	
Q		51	(81)	
R		52	(82)	
S		53	(83)	
T		54	(84)	
U		55	(85)	
V		56	(86)	
W		57	(87)	
X		58	(88)	
Y		59	(89)	
Z		5A	(90)	
[		5B	(91)	
\		5C	(92)	
]		5D	(93)	
^	↑	5E	(94)	
_	←	5F	(95)	
`		60	(96)	
a	♠	61	(97)	
b		62	(98)	
c		63	(99)	
d		64	(100)	
e		65	(101)	
f		66	(102)	
g		67	(103)	
h		68	(104)	
i		69	(105)	
j		6A	(106)	
k		6B	(107)	
l		6C	(108)	
m		6D	(109)	
n		6E	(110)	
o		6F	(111)	
p		70	(112)	
q	●	71	(113)	
r		72	(114)	
s	♥	73	(115)	
t		74	(116)	
u		75	(117)	
v	☒	76	(118)	
w		77	(119)	
x	♣	78	(120)	
y		79	(121)	
z	♦	7A	(122)	
{		7B	(123)	

(*continued*)

Character		Value	
Std.	64	hex	(dec)
¦		7C	(124)
¦		7D	(125)
OVLN		7E	(126)
DEL		7F	(127)
	Orange	81	(129)
	f1	85	(133)
	f3	86	(134)
	f5	87	(135)
	f7	88	(136)
	f2	89	(137)
	f4	8A	(138)
	f6	8B	(139)
	f8	8C	(140)
		8D	(141)
		8E	(142)
		90	(144)
		91	(145)
		92	(146)
		93	(147)
		94	(148)
	Brown		(149)
	Lt. Red		(150)
	Grey 1		(151)
	Grey 2		(152)
	Lt. Green		(153)
	Lt. Blue		(154)
	Grey 3		(155)
		9C	(156)
		9D	(157)
		9E	(158)
		9F	(159)
		A0	(160)

Values		Same as:	
C0–DF	(192–223)	60–7F	(96–127)
E0–FE	(224–254)	A0–BE	(160–190)
FF	(256)	7E	(126)

Character		Value	
Std.	64	hex	(dec)
		A1	(161)
		A2	(162)
		A3	(163)
		A4	(164)
		A5	(165)
		A6	(166)
		A7	(167)
		A8	(168)
		A9	(169)
		AA	(170)
		AB	(171)
		AC	(172)
		AD	(173)
		AE	(174)
		AF	(175)
		B0	(176)
		B1	(177)
		B2	(178)
		B3	(179)
		B4	(180)
		B5	(181)
		B6	(182)
		B7	(183)
		B8	(184)
		B9	(185)
		BA	(186)
		BB	(187)
		BC	(188)
		BD	(189)
		BE	(190)
		BF	(191)

D

THE INSTRUCTION SET

This appendix contains summaries of the effects and addressing mode options of all 6510 instructions. Beside each addressing mode name is a set of braces enclosing two values: the opcode for that combination of operation and addressing mode, and the execution time of the complete instruction in microseconds. An asterisk beside an execution time indicates that the instruction will take an additional microsecond to execute if it lies across a 100h page boundary. A "+1" beside a branch indicates an additional microsecond if the branch is taken.

The symbols used in the instruction-effects summaries are as follows: A = accumulator, P = status register, X = X register, Y = Y register, PC = program counter, S = stack pointer, TOS = top-of-stack location pointed to by S, M = memory location, INDX = program-supplied 8-bit 2C-index value, d(i) = bit *i* of accumulator or memory location (depending on addressing mode), ADDRESS = address in or pointed to by operand, (FFFE,FFFF) = the contents of locations FFFE and FFFF, \ = thrown away, and the flags; Z = zero flag, N = negative flag, C = carry flag (C with an overbar means NOT carry), O = overflow flag, B = break flag, D = BCD-mode flag, and I = interrupt-inhibit flag.

For more detailed explanations of the 6510 instructions, see Chapter 3.

Operation	Direct modes	Indexed modes	Flags
ADC [A + M + C → A]	Absolute {6D,4} Zero-page {65,3} Immediate {69,2}	Absolute,X {7D,4*} Absolute,Y {79,4*} Zero-page,X {75,4} Indirect indexed {71,5*} Indexed indirect {61,6}	Z N C O
AND [A AND M → A]	Absolute {2D,4} Zero-page {25,3} Immediate {29,2}	Absolute,X {3D,4*} Absolute,Y {39,4*} Zero-page,X {35,4} Indirect indexed {31,5*} Indexed indirect {21,6}	Z N
ASL [d(i) → d(i + 1) ; d7 → C,0 → d0]	Absolute {0E,6} Zero-page {06,5} Implied {0A,2}	Absolute,X {1E,7} Zero-page,X {16,6}	Z N C
BCC [IF C = 0, THEN PC + 1 + INDEX → PC]		Relative {90,2 + 1*}	None
BCS [IF C = 1, THEN PC + 1 + INDEX → PC]		Relative {B0,2 + 1*}	None
BEQ [IF Z = 1, THEN PC + 1 + INDEX → PC]		Relative {F0,2 + 1*}	None
BIT [A AND M → Z, d6(M) → V, d7(M) → N]	Absolute {2C,4} Zero-page {24,3}		Z N O
BMI [IF N = 1, THEN PC + 1 + INDEX → PC]		Relative {30,2 + 1*}	None
BNE [IF Z = 0, THEN PC + 1 + INDEX → PC]		Relative {D0,2 + 1*}	None
BPL [IF N = 0, THEN PC + 1 + INDEX → PC]		Relative {10,2 + 1*}	None
BRK [PC + 2 → TOS, S + 1 → S, (FFFE,FFFF) → PC]	Implied {00,7}		B I
BVC [IF V = 0, THEN PC + 1 + INDEX → PC]		Relative {50,2 + 1*}	None

Operation	Direct modes	Indexed modes	Flags
BVS [IF V = 1, THEN PC + 1 + INDEX → PC]		Relative {70,2 + 1*}	None
CLC [0 → C]	Implied {18,2}		C
CLD [0 → D]	Implied {D8,2}		D
CLI (enable ints) [0 → I]	Implied {58,2}		I
CLV [0 → V]	Implied {B8,2}		O
CMP [A − M → \ ]	Absolute {CD,4}	Absolute,X {DD,4*} Absolute,Y {D9,4*}	Z N
	Zero-page {C5,3}	Zero-page,X {D5,4} Indirect indexed {D1,5*} Indexed indirect {C1,6}	C
	Immediate {C9,2}		
CPX [X − M → \ ]	Absolute {EC,4} Zero-page {E4,3} Immediate {E0,2}		Z N C
CPY [Y − M → \ ]	Absolute {CC,4} Zero-page {C4,3} Immediate {C0,2}		Z N C
DEC [M − 1 → M]	Absolute {CE,6} Zero-page {C6,5}	Absolute,X {DE,7} Zero-page,X {D6,6}	Z N
DEX [X − 1 → X]	Implicit {CA,2}		Z N
DEY [Y − 1 → Y]	Implicit {88,2}		Z N
EOR [A XOR M → A]	Absolute {4D,4}	Absolute,X {5D,4*} Absolute,Y {59,4*}	Z N
	Zero-page {45,3}	Zero-page,X {55,4} Indirect indexed {51,5*} Indexed indirect {41,6}	
	Immediate {49,2}		
INC [M + 1 → M]	Absolute {EE,6} Zero-page {E6,5}	Absolute,X {FE,7} Zero-page,X {F6,6}	Z N
INX [X + 1 → X]	Implicit {E8,2}		Z N
INY [Y + 1 → Y]	Implicit {C8,2}		Z N
JMP [ADDRESS → PC]	Absolute {4C,3} Absolute- indirect {6C,5}		None

(*continued*)

Operation	Direct modes	Indexed modes	Flags
JSR [PC+2→TOS; ADDRESS→PC]	Absolute {20,6}		None
LDA [M→A]	Absolute {AD,4} Zero-page {A5,3} Immediate {A9,2}	Absolute,X {BD,4*} Absolute,Y {B9,4*} Zero-page,X {B5,4} Indirect indexed {B1,5*} Indexed indirect {A1,6}	Z N
LDX [M→X]	Absolute {AE,4} Zero-page {A6,3} Immediate {A2,2}	Absolute,Y {BE,4*} Zero-page,Y {B6,4}	Z N
LDY [M→Y]	Absolute {AC,4} Zero-page {A4,3} Immediate {A0,2}	Absolute,X {BC,4*} Zero-page,X {B4,4}	Z N
LSR [d(i)→d(i−1) ; 0→d7, d0→C]	Absolute {4E,6} Zero-page {46,5} Implied {4A,2}	Absolute,X {5E,7} Zero-page,X {56,6}	Z N C
NOP	Implied {EA,2}		None
ORA [A OR M→A]	Absolute {0D,4} Zero-page {05,3} Immediate {09,2}	Absolute,X {1D,4*} Absolute,Y {19,4*} Zero-page,X {15,4} Indirect indexed {11,5*} Indexed indirect {01,6}	Z N
PHA [A→TOS, S+1→S]		Stack {48,3}	None
PHP [P→TOS, S+1→S]		Stack {08,3}	None
PLA [S−1→S, TOS→A]		Stack {68,4}	Z N
PLP [S−1→S, TOS→P]		Stack {28,4}	All
ROL [d(i)→d(i+1) ; d7→C, C→d0]	Absolute {2E,6} Zero-page {26,5} Implied {2A,2}	Absolute,X {3E,7} Zero-page,X {36,6}	Z N C
ROR [d(i)→d(i−1) ; C→d7, d0→C]	Absolute {6E,6} Zero-page {66,5} Implied {6A,2}	Absolute,X {7E,7} Zero-page,X {76,6}	Z N C
RTI [TOS→P, S+1→S, TOS→PCL, S+1→S, TOS→PCH S+1→S]	Implied {40,6}		All

Operation	Direct modes	Indexed modes	Flags
RTS [TOS→PCL, S+1→S, TOS→PCH, S+1→S, PC+1→PC]	Implied {60,6}		None
SBC [A−M−$\overline{C}$→A]	Absolute {ED,4}	Absolute,X {FD,4*} Absolute,Y {F9,4*}	Z N
	Zero-page {E5,3}	Zero-page,X {F5,4} Indirect indexed {F1,5*} Indexed indirect {E1,6}	C O
	Immediate {E9,2}		
SEC [1→C]	Implied {38,2}		C
SED [1→D]	Implied {F8,2}		D
SEI (disable ints) [1→I]	Implied {78,2}		I
STA [A→M]	Absolute {8D,4}	Absolute,X {9D,5} Absolute,Y {99,5}	None
	Zero-page {85,3}	Zero-page,X {95,4} Indirect indexed {91,6} Indexed indirect {81,6}	
STX [X→M]	Absolute {8E,4} Zero-page {86,3}	 Zero-page,Y {96,4}	None
STY [Y→M]	Absolute {8C,4} Zero-page {84,3}	 Zero-page,X {94,4}	None
TAX [A→X]	Implied {AA,2}		Z N
TAY [A→Y]	Implied {A8,2}		Z N
TSX [S→X]	Implied {BA,2}		Z N
TXA [X→A]	Implied {8A,2}		Z N
TXS [X→S]	Implied {9A,2}		None
TYA [Y→A]	Implied {98,2}		Z N

JAMES SUTTON

POWER PROGRAMMING THE COMMODORE 64

ASSEMBLY LANGUAGE, GRAPHICS AND SOUND

The Complete Programmer's Guide

Preface: Exposé. . . Past Its Plastic Envelope: The Computer's Inner Machinery. . . Conceptual Quicksilver: Data Structures. . . Into Its Brain: 6510 Assembly Language. . . Imposing Reason: Program Planning. . . Connecting the Nerves: Using the Memory Map. . . Awakening the PIXY: Advanced Graphics. . . Vocal Chords for a Chimera: Sound Synthesis

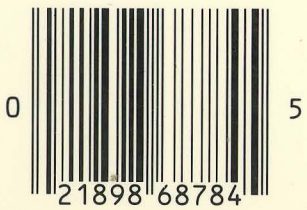
In these stimulating chapters you will find a complete method for translating your most ambitious Commodore 64 programming inspirations into reality. Sparkling style and paced presentation work together to make it easy for you to master the most advanced assembly language, graphics, sound, and I/O techniques. You benefit from the author's extensive microcomputer programming experience, and the many 'trade secrets' obtained from top professional Commodore 64 programmers and revealed here for the first time. Whether for a tutorial or a text, reference or recreation, Power Programming is the first and last computer book your library will need.

"I have yet to read a technical discussion that has been so thoroughly "milled" into such easily digestible form. His descriptions. . . held me absolutely spellbound. An incredible treatise of theoretical and experiential knowledge. Quite remarkable in light of the sophistication of the subject material. I'd recommend the book. . . even to my best friends."

Jack Clarke, Professor, El Camino Community College District

"I am particularly impressed with his style and his method of presentation. The author seems to have many unique qualities in his ability to attack the subject. He's done an excellent job of explaining those 'sometimes overlooked' details. A well organized, detailed account of the Commodore 64, designed for a wide range of audience. An excellently executed document covering the Commodore 64 in more detail than any other document I've seen."

Ed Pevovar, President, Computer Communication and Engineering Consultants



PRENTICE-HALL, INC.
Englewood Cliffs, N.J. 07632

ISBN 0-13-687849-0