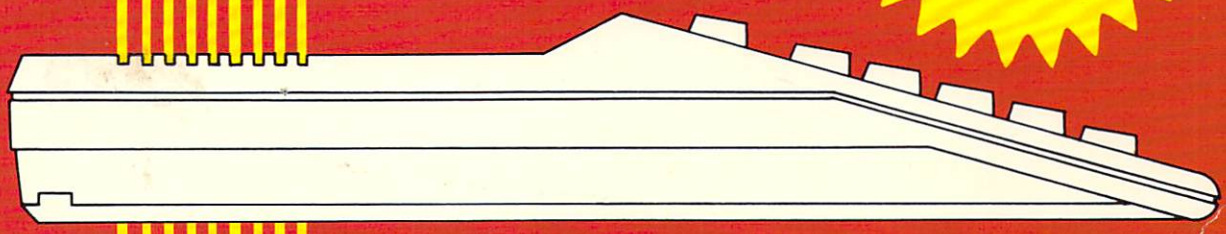
2732

COMMODORE 128™ BASIC PROGRAMMING TECHNIQUES

MARTIN HARDEE

COVERS COMMODORE BASIC, VERSION 7.0!



COMMODORE 128™ BASIC PROGRAMMING TECHNIQUES

MARTIN HARDEE

TAB TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

This book is dedicated to my mother, who will never understand computers;
to my father, who is trying; and to my wife, Jan, who routinely puts up with four or five in the house at one time.

FIRST EDITION
FIRST PRINTING

Library of Congress Cataloging in Publication Data

Hardee, Martin.
Commodore 128 BASIC.

Includes index.
1. Commodore 128 (Computer)—Programming.
2. BASIC (Computer program language) I. Title.
QA76.8.C645H37 1986 005.265 86-14373
ISBN 0-8306-0432-4
ISBN 0-8306-2732-4 (pbk.)

Contents

Introduction

A computer can change your life . . .
. . . or it can sit in the corner gathering dust.

All too many of us bought our machines with high hopes, only to have those hopes dashed by complicated instruction manuals, high-priced software, and a general feeling of helplessness.

We all know there's a tremendous amount of power inside your Commodore 128. The secret is harnessing it, and harnessing it without delving into the realms of memory swaps, assembly language, or complicated and confusing peeks and pokes. There's no assembly language in this book. It's all pure BASIC. In addition, the routines in these chapters are *structured* and *modularized*, so they can be transplanted into any BASIC program you write.

This book will dispel many of the myths you've heard and allay many of the fears you have about do-it-yourself programming. Chapter 1 begins by examining the new commands and features available under BASIC 7.0. These new commands will give you a fresh way of looking at the tiring old BASIC language.

In Chapter 2, you'll see how to take advantage of the Commodore's memory setup, examine how graphic screens are handled, and learn a few tricks about the keyboard (it's not quite as simple as typing a letter).

Chapter 3 zeroes in on *devices*. You'll learn how to communicate with the disk drives, modems, printers, monitors, and other *peripherals* that can be connected to the Commodore 128. The simple concepts in this chapter provide the building blocks for file handling, printing, and other common operations.

One of the most important applications in computing is the ability to search for information. Chapter 4 shows how to design simple BASIC routines that will find the information you're looking for, whether it's on disk or in memory.

In Chapter 5, you'll see how to create programs that save and recall streams of data to and from diskette. The modules in this chapter can be easily modified and inserted into any program that needs to save data.

Chapter 6 focuses on *relative files*—the filing

method most often used to store names, addresses, accounting data, and other business information.

In Chapter 7, you'll see how to screen and trap characters as they are entered from the keyboard, giving you complete control over operator entry. There are even routines for performing auxiliary operations (playing music and displaying the time) while awaiting input.

Sorting, one of the most important computer concepts of all, is discussed in Chapter 8. There you will find detailed examples of three different sorting techniques.

The appearance of today's software is radically different from programs being written just a few years ago. Bar menus, screen entry forms, and other *user friendly* devices have become standard—it's seldom you'll see quality software written without them. Chapter 9 shows how to dress up your programs using these features, with a little fuss.

BASIC's Achilles heel is speed: if a BASIC program isn't written properly, it can grind along at the speed of a city bus. Chapter 10 shows how to keep your programs running at peak efficiency, while avoiding some of the pitfalls that come from over-optimizing programs (lack of readability, etc.).

Chapter 11 explores the "bullet-proofing" of your programs and steers you around some of the potholes involved in trapping errors.

Graphics, the most dazzling feature of the Commodore 128, are covered in Chapters 12 and 13. You'll see how to create, position and color pictures, and how to design and animate shapes using built-in BASIC 7.0 commands.

We end up with a discussion on music and sound. Chapter 12 concentrates on how to play music and harmonies. It includes a brief discussion of creating sound effects. You'll also see references to the C-128's music and sound commands throughout the text.

Again, all of the program routines in this book are self-contained, so they can generally be mixed and matched at will. You'll no doubt be using these concepts in this book in every new program you write. And you may well find yourself going back to old programs and enhancing them with these new features.

Chapter 1

A Refreshing Approach to BASIC

Slick programming used to be a tricky business. Sound and pictures—the ingredients that make even business programs enjoyable—were no joy at all to produce.

If you wanted graphics, you wrote complicated routines in assembly language—and then spent hours trying to *debug* them. Getting a frog to realistically devour a mosquito could take days.

Sound was just as complicated. Even today, getting two or three instruments to play in harmony is next to impossible on many computers.

Disk storage, the recall of information such as names and addresses, could be had, but the price was hours spent pouring over complicated manuals that only hinted at the proper procedures.

With the introduction of powerful, less expensive computers like the Commodore 128 many of these frustrations vanish. First, operations that used to take hours to program by hand are now incorporated into BASIC commands; if you know the tricks, you can now draw a perfect circle (or ellipse, or rectangle) in a matter of seconds. Getting a frog to devour a mosquito is no longer so complicated. It can be done right from BASIC, using sprites and the aid of the Commodore 128's built-in picture editor.

And while data-handling operations such as name and address storage are still no picnic, new commands make these operations a good deal easier to comprehend.

A MAGICIAN'S TRADE

There's a lot of illusion built into computer programs. In sophisticated programs, just as in sophisticated magic tricks, most of the work takes place out of view. Once you know the secrets, however, programming, like performing novelty magic tricks, is easy. Unfortunately, you won't find most of these secrets in your manual, because manuals are written by programmers who assume everyone already knows all the tricks.

This book will unravel the secrets of Commodore 128 BASIC. You'll learn how to store and recall data, electronically search through information, design great graphics, produce rousing musical cho-

ruses, and create the kind of professional-looking software that has been difficult to come by on reasonably-priced computers.

We'll pay particular attention to new BASIC commands that let programs run faster and make them much easier to understand. Areas that include traditionally technical subjects are written in plain English and are richly illustrated.

A WALK DOWN MEMORY LANE

The Commodore 128 does a lot of things other computers don't. Even when it's performing mundane processing tasks, it does so more quickly and elegantly than comparable machines.

Before we peer into the full capabilities of BASIC 7.0, perhaps it is a good idea to look into the past for a moment. The program listed in Fig. 1-1 would run on most of today's popular microcomputers. Lines 10 and 20 are comments to explain what the program does. When the program is run, line 30 accepts the users name and stores it in a variable called NA$. Because names contain letters, a dollar sign ($) follows the NA, identifying it as a *string* variable. Line 40 prints a line space on the screen, and line 50 displays a message that incorporates the user's name. Not until line 60 do we really get down to business. The GOSUB 200 statement instructs the computer to perform a separate subprogram, or *subroutine*, which begins at line 200. This subroutine is, in fact, the core of the whole program, because it creates and prints the random numbers. When ten random numbers are generated, the FOR. . .NEXT loop is complete, and the computer RETURNs from the subroutine. The program wraps things up by asking if the user desires another listing. Line 80 accepts the response, and 85 makes Y$ equal to the leftmost character of that response (translating, in effect, YES into Y, and NO into N). Only if the system receives a Y will it branch back to line 60 and print more numbers. Otherwise, a farewell message is displayed and the end statement at line 110 halts the program. If any of this seems foreign, you'll probably want to study Appendix A, "A Refresher Course in BASIC." It's located at the back of this book.

```
10  REM  Personal Random Number Generator
20  REM
30  INPUT "WHAT IS YOUR NAME ?";NA$
40  PRINT
50  PRINT "HELLO ";NA$;"! HERE ARE 10 RANDOM NUMBERS:"
60  REM  Now Generate Numbers
70  GOSUB 200
80  INPUT "VIEW ADDITIONAL NUMBERS (Y/N)?";Y$
85  Y$=LEFT$(Y$,1)
90  IF Y$="Y" OR Y$="y" THEN GOTO 60
100 PRINT "THANK YOU FOR AN ENJOYABLE GENERATION"
110 END
200 REM  Generate and Print Number
210 FOR X=1 TO 10
220   R=RND(0)*10
230   PRINT R
240 NEXT
250 RETURN
```

Fig. 1-1. Personal random number generator: an example of a program that could be written in most dialects of BASIC. It is workable, but hard to read.

```
10  REM  Improved Random Number Generator
20  :
30  INPUT "WHAT IS YOUR NAME ?";NAME$
40  PRINT
50  PRINT "HELLO "NAME$"! HERE ARE 10 RANDOM NUMBERS:"
60  REM  Now Generate Numbers
65  DO UNTIL Y$="N"
70    GOSUB 200
80    INPUT "VIEW ADDITIONAL NUMBERS (Y/N)?";Y$
85    Y$=LEFT$(Y$,1)
90  LOOP
100 PRINT "THANK YOU FOR AN ENJOYABLE GENERATION"
110 END
120 :
200 REM  Generate and Print Number
205 AT=0:LAST=10
210 DO UNTIL AT=LAST
215   AT=AT+1
220   R=RND(0)*10
230   PRINT USING " ### #.##";R
240 LOOP
250 RETURN
```

Fig. 1-2. Improved random number generator, an example of a program written in Commodore 128 BASIC 7.0. It is more elegant, easier to read, and it works just as well.

WHAT MAKES BASIC DIFFERENT ON THE C-128

The Commodore 128 would have no trouble at all interpreting the BASIC listing in Fig. 1-1; you could turn on the computer, type in the program, and be assured that it would run the first time through. There are, however, better ways to do things on the Commodore 128.

The most obvious change might involve the "view additional numbers" section of the program. Even though you're familiar with BASIC, doesn't it seem complicated? The program has to test the user's entry each time through, and then make a decision as to whether or not to branch back to a specific line number.

It's as if your mail carrier knocked on the door every day and asked "Do you want me to deliver your mail to the same address tomorrow, too?" Instead, the post office generally assumes you want your mail delivered to the same place until it's notified that you're moving. It's a much simpler arrangement.

Commodore 128 BASIC 7.0 offers a set up similar to the post office's called DO UNTIL. In the Commodore 128 BASIC rewrite of the Random Number Generator in Fig. 1-2, you can get a good idea of how it works.

The program still checks with the user each time to see if another random number listing is necessary, but the rest is automatic. The computer has been told to perform all operations between the DO UNTIL and the LOOP until Y$ is equal to N. In this case, there are only two lines between the two statements: There is line 70, which *calls* the numbers display routine at line 200, and there is line 80, which asks if the user would like another go. There can be as many statements in a loop as you like, but it's most effective to design small loops

so you can clearly see the beginning and end.

Using DO/UNTIL Instead of FOR . . . NEXT

We're not finished with Random Number Generator just yet. Take a look at the heart of the program, the "compute and display" routine at line 200. For the sake of illustration, we've replaced the FOR . . . NEXT loop with another form of DO . . . WHILE structure. In this case, the variable AT keeps track of whether the program is printing the first random number, the second, or the third and so on. AT works like an amusement park turnstile, adding one to itself with each pass. When AT finally equals LAST (in this case, 10) the loop ends.

Granted, this DO . . . UNTIL loop contains one statement more than the original FOR . . . NEXT loop presented in the first listing (the statement which bumps up the value of AT), but notice how much easier the DO . . . UNTIL structure is to read. It's almost as if it were written in English!

Formatting Numbers

Another change we've made is to the statement that prints the random numbers (line 230). If you've worked with numeric variables on another computer, you know they're often displayed to the ninth or tenth decimal place. It's great for engineers, but it gives most other folks a case of eye strain. People usually like to view numbers rounded to two or three decimal places.

Commodore 128 BASIC's PRINT USING statement does all the rounding automatically, and prints the numbers so they'll stack up into nice neat columns:

Without PRINT USING	With PRINT USING
2.3432	2.34
3.1263	3.12
121.9723	121.97

Pound signs (#) in the PRINT USING statement represent the numeric format desired. You can even include dollar signs, commas, and other symbols to make the format suitable for checks and financial statements:

```
PRINT USING "$#,###.##";1234.567
```

You could use the above statement, for instance, to display 1234.567 in a proper dollar format ($1,234.57).

BASIC 7.0 is chock full of statements like this that are guaranteed to make your programming less tedious and your programs more professional. Let's take a look at another one.

Inside a String

Any professionally designed program would screen keyboard entry far better than either of our original listings have done. What would happen if a user accidentally typed T instead of Y at the "VIEW ADDITIONAL NUMBERS" question? Or simply pressed return without typing anything at all?

In both cases, the program would display a pleasant goodbye message and end itself. This sort of unexpected default is best to avoid because it makes users feel they can't make a mistake. If they do, they could get dumped out of the program all together. It's a situation that makes users insecure—something that should be avoided at all costs!

An old method for tackling this sort of problem would have been to add new logic after line 85 which would screen out any invalid response:

```
80  INPUT "VIEW ADDITIONAL NUMBERS (Y/N)?";Y$
85  Y$=LEFT$(Y$,1)
87  IF Y$< >"Y" OR Y$< >"y" OR Y$< >"N" OR Y$< >"n" GOTO 80
```

While line 87 indisputably redisplays the question when Y$ isn't equal to a valid response, it's a little tedious—tedious not only to type in originally, but to decipher later when reviewing the printed listing.

Here's a better way, again using a loop structure called DO/WHILE and a new statement called INSTR (instring):

```
74  Y$=" "
76  DO WHILE INSTR("YyNn"$)=0
```

```
80      INPUT "VIEW ADDITIONAL
        NUMBERS (Y/N)?";Y$
85      Y$=LEFT$(Y$,1)
87      LOOP
```

The INSTR function simply presents a *string* of characters (In this case, "YyNn") and determines if Y$ matches all or part of it. If there's a match, the INSTR function returns the exact location. For example, if Y$ equals "y", the value is two (because lowercase "y" is in the second position of the string "YyNn"). In any event, if Y$ is equal to any of the characters in the first string, the value returned by INSTR will be greater than zero, and the loop will end.

See the INSTR chart in Fig. 1-3 for further examples.

The only time the value of INSTR("YyNn",Y$) equals 0 is when no match is found. This function lets the computer screen out invalid keyboard entry without resorting to contorted statements like those in the previous example. Note that in line 74 it was necessary to "blank out" (*initialize*, in computer jargon) Y$ so that there is no possibility of Y$ being equal to a valid response before the loop starts. You'll notice the same type of *initialization* with the numeric variable AT at line 205 in the other DO/UNTIL loop, where we make AT=0 before starting. The reason for blanking these variables is simple: If Y$ or AT were valid at the beginning (if AT=10 or Y$="Y"), the program would branch down to the end of the loop immediately, bypassing the loop completely! Even though BASIC blanks out variables each time a program is run, it's a good practice to initialize any variables that will be used in routines repeatedly, such as those in DO . . . WHILE loops.

WHAT'S AHEAD

So far we've only examined one very simple program, but already you should be getting a feel for the power of Commodore 128 BASIC 7.0. This first chapter has presented commands that:

- Make program routines easier to read and understand
- Print rounded numbers, formatted into columns
- Search for sets of characters within other character strings.

The Commodore 128's INSTR command may be used anytime you wish to look for characters. The most common applications are searching and input screening routines. The command's format, INSTR(A$,B$), translates as "search the contents of A$ for the group of characters contained in B$." Naturally, either of the strings may be a literal group of characters enclosed in quotes, such as "Hello" or "YyNn". If no match is found, INSTR(A$,B$) equals 0. If there is a match, this function returns the position where the match was found:

Value Returned	Y$	
INSTR("YyNn",Y$)=1	"Y"	(Y is the 1st position)
INSTR("YyNn",Y$)=2	"y"	(y is the 2nd position)
INSTR("YyNn",Y$)=3	"N"	(N is the 3rd position)
INSTR("YyNn",Y$)=4	"n"	(n is the 4th position)
INSTR("YyNn",Y$)=0	no match	(Anything other than what's contained in the first item)

This particular INSTR test would be a fast and efficient way of checking the user's answer to a question that should only have a yes or no answer. See the text for a program example.

Fig. 1-3. Searching and testing the easy way.

Of course, there's a lot more that we could have added to this program. For example, a routine to play random notes could be easily incorporated. We could have added a routine to produce circles and boxes as a visual accompaniment to the sound routine. We could even have added a few lines that would save the result of our random number generation in a file on disk.

In the remainder of this book, we'll explore these additional commands and examine how to put them to practical application.

Chapter 2

What You Have to Work With

The Commodore 128 offers many features that make it attractive to programmers. It can function as three different computers; it produces both 40- and 80-column displays; it gives you an ample amount of available memory; and it offers a host of features that simplify the chore of program editing.

THE THREE SIDES OF THE COMPUTER

As you've probably already found out from your manual, and from the advertising sales pitch that preceded your purchase, the Commodore 128 is actually three computers in one package. Commodore likes to refer to these computers as different *sides* of the machine. Figure 2-1 lists the advantages and disadvantages of each side of the C-128.

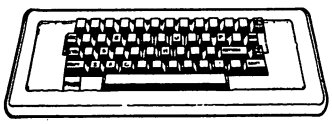
The 64 Side

There's the tried and trusted Commodore 64 side, which runs all programs ever written for this versatile work horse of a computer. If you start the Commodore 128 while a video game or other cartridge program is installed, the system enters the C-64 side of the system automatically. Programs from cassette or disk can be run by typing GO 64 from the Commodore 128 side of the machine, and then following the loading procedure outlined in the program's manual. All of these programs work exactly as they would on a Commodore 64. The same is true for the BASIC and assembly language program listings you see in magazines and books.

But the Commodore 64 doesn't give much bang for the buck when it comes to BASIC commands. In fact, once you become accustomed to the Commodore 128 side of the machine, you may do no programming at all on the C-64 side.

The 128 Side

This is the part of the system we'll be dealing with most extensively in this book. In addition to a speedier and more versatile BASIC, the 128 side of the machine has twice the memory of the already amply endowed Commodore 64. Disk operations



Pluses and Minuses of their operation

The Commodore 64 Side

+ Runs all C-64 software
+ Operates all C-64 devices including printers, monitors, joysticks, and controllers.
- Less memory than the C-128 side of the machine
- Operates with a less expanded BASIC than the C-128 side.

The Commodore 128 Side

+ Twice the memory of the C-64 side of the machine
+ Faster and more versatile BASIC
+ Additional keyboard functions
- Machine language programs for the C-64 generally won't run on this side

The CP/M Side

+ Runs many popular CP/M programs, such as WordStar, Perfect Writer, and Perfect Calc
+ Reads CP/M disks originally created on other machines such as KAYPRO and Xerox
- Most commands are different form those of the C-64 and C-128 sides
- Disks from CP/M side cannot be read by the C-64 and C-128 sides

Fig. 2-1. The three computers inside your Commodore 128.

are also much easier to accomplish on the 128 side of the machine, because disk commands more nearly resemble English words.

C-64/C-128 Compatibility

As you experiment, you'll find that many BASIC programs written for the Commodore 64 will run perfectly on the Commodore 128 side of the machine. After you've been suitably spoiled by C-128 BASIC 7.0, however, you'll find yourself spicing up these older programs to take advantage of the added power on the 128 side.

Machine language or *assembly language* programs written for the Commodore 64 generally won't run on the 128 side of the system, because many memory locations have changed. The same is true for BASIC programs that use lots of PEEK and POKE statements, although some PEEKs and POKEs are still the same as on the Commodore 64.

Remember again that any program written for a Commodore 64 will run on the C-64 side of the system, regardless of whether it runs on the 128 side.

CP/M Plus

The third part of the Commodore 128 is com-

prised of a processor designed to run thousands of available programs written for the CP/M operating system. Like the 128's built-in disk operating systems, CP/M has commands for seeing what's on a disk, making copies, and running programs. But here's the catch: All of the commands are completely different. Commands that work on the 128 side of the system will have no effect whatever on the CP/M side (except to produce an error message). The CP/M Plus section of the Commodore 128 is like operating a completely different computer. The commands are different. Even the procedures are different. For example, BASIC is always available for use on the 64 and 128 sides of the machine—it's built into the system. But when running under CP/M, BASIC must be loaded from a CP/M disk. And many of the BASIC commands are different from what you'd find in C-64 BASIC or in the C-128 side's BASIC 7.0.

Even with all of this confusion and incompatibility, the CP/M side of the Commodore 128 still offers splendid advantages to those interested in industrial-strength business software. Some of the most versatile word processing, filing, and accounting software packages around today have been written for computers using CP/M. And this software runs on the Commodore just as well as it does on more expensive business systems. Besides, what other CP/M system lets you play Donkey Kong after you've finished with Perfect Writer or WordStar?

COMMODORE 128 MEMORY

There's a saying in computerdom that "any program will expand to devour all available memory." Still, it is hard to imagine any BASIC program that would completely fill up the 128's electronic gas tank.

The 128 side of your Commodore 128 has a staggering amount of memory: As shown in Fig. 2-2, when you turn on the machine, more than 58-thousand bytes (characters) are available to hold your program alone. (Just to give you an idea, a 40K program would contain almost 2,000 lines—and you'd still have 18K free for more program!)

If you use graphics, plan on having about 9K less storage area for your program; when you use the high-resolution or multicolor graphics modes, the Commodore 128 automatically takes over this amount of memory for screen display. Incidentally, as you'll see on the section covering graphics, a simple command is used for entering and leaving the graphics modes. There's seldom any need to worry about complex memory management with the Commodore 128 because BASIC handles these internal affairs automatically.

Variables are stored in a special section of memory and can take up another 64K. That's enough space to store a list of more than 1,300 items and still have room for other information.

The amount of variable space available is not affected by the size of your program. In fact, the program and variables are stored in two separate *banks* of memory.

What Memory and Power Mean to Programmers

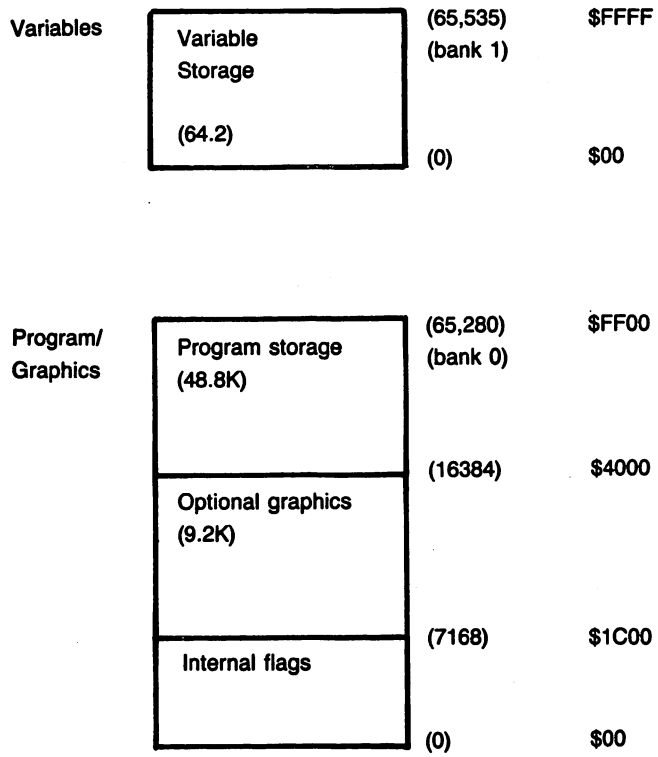
If you've programmed in BASIC on other computers, it may be necessary to readjust your thinking. First, operations you may have split into several programs (say, printing and sorting) can now be combined in one program. You can also add program features without worrying about running out of memory. More space means you can spread out program statements that you might be tempted to squeeze into one line on lesser machines. This statement:

```
10 FOR X=1 TO 10: PRINT "TEST #
   ";X;:NEXT:PRINT
```

can now be typed as:

```
10 FOR X=1 TO 10
20 PRINT "TEST #";X;
30 NEXT
40 PRINT
```

Additional variable memory means not only more storage space, but also faster operation for many programs, because information may be stored in memory rather than being written and read con-



BASIC. Each line of BASIC begins as the lowest available memory location. Thus, lines are stored sequentially in memory. When you add a line to an existing program, BASIC pushes all higher-numbered lines up in memory to make room for your new one. This is why the machine will pause for a second when you press return after adding a line to a very large program (generally noticeable at 20K or more).

VARIABLES. Variables are stored in a separate *bank* of the 128's memory. The computer always knows that these variables are in this upper bank, so there's no need for most BASIC programmers to be concerned with bank switching.

String variables are stored from the top of memory down, and numeric variables are stored from the bottom up. When the two sides meet, memory is *full* and an "out of memory" message will be displayed.

Fig. 2-2. How memory is broken up.

tinually from disk.

A good example is a name and address program. On computers with more limited memory, you might store each address separately on disk, using *random access* files. This requires more programming work and processing time than the alternative, which is to store all names and addresses in memory, reading them only at the beginning of the program, and saving them before exiting. Because the Commodore 128's expanded memory can store hundreds of names and addresses in memory at one time, access to the data is instant, meaning

faster program operation.

Variable Memory

Because variable memory is separate from program memory, the size of your program won't affect the amount of variable work space available. On many computers, large programs cut down on the amount of memory available for storing and processing. But the Commodore 128 allows you to create very large programs without compromising your variable work space.

A Better Type of Memory

Variables in memory are cleared in only three ways:

1. When you type NEW to clear the current program from memory.
2. When you RUN another program.
3. When you turn off the power to the C-128.

Note that variables aren't erased when you add program lines! It's a little thing, perhaps. But it's a contrast to the way most other computers operate.

Most other BASIC's have been designed so that adding or even editing a program line would wipe out all variables stored in memory at the time. So when you were trying to debug a program, you couldn't change a line and resume running, because all data in memory would be gone!

Because the C-128 retains all variables during program editing, you're free to make changes to a line and resume the program with a GOTO and a line number.

The Commodore 128's memory arrangement provides phenomenal freedom for developing your programs under BASIC.

GRAPHIC SIDES OF THE COMPUTER

Undoubtedly one of the most exciting elements of the Commodore 128 is that of video display. It is also one of the most confusing. The C-128 boasts four graphic screens and several graphic modes: there's your eighty column text. There's your forty column text. There's a high-resolution graphics screen, which allows you to design pictures of very high detail and limited color. Or there's multicolor high-resolution screen, which allows pictures of moderate detail and allows more color. To confuse things even more, you can have screens that are half graphics and half text, and you can change the number of lines in this mix of text and graphics. There's even a way to mix characters and graphics throughout the screen. Figure 2-3 illustrates and explains these modes.

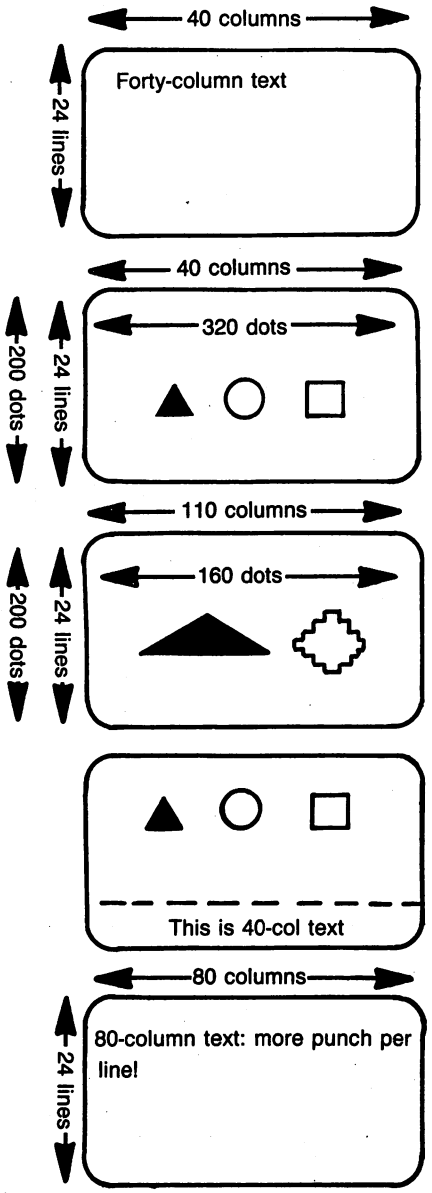
It's all very confusing, even when you've worked with the system for a while. But reward for sticking to it and learning about the ins and outs of these graphics modes is that you'll be able to greatly enhance your programs with professional graphics.

The 40-Column Side

If you're using a television set as a monitor, or if you bought your computer for both games and business, chances are your system is operating in a forty column mode. Television sets and forty column monitors display 40-characters-per-line text and graphics from both high-resolution graphics modes. Sound is delivered either through the monitor speaker or the speaker in your television set. Surprisingly, the sound from a TV set is every bit as good as sound from the more expensive monitor. But the picture quality of 40-column monitors is definitely crisper and brighter. After all, computer monitors were designed expressly for computers, and that's where they shine.

The 80-Column Side

The eighty-column side of the machine is another world entirely. In fact, the Commodore eighty-column monitor even has its own plug, labeled RGBI (it stands for red, green, blue, and indigo—a representation of the main color combinations being *ported* to this monitor). The eighty-column display is especially well-suited for word processing, since it allows typists to view a letter exactly as it will appear on the printed page (see Fig. 2-4). Forty-column word processors generally arrange text automatically and don't really allow



The forty-column mode, native to the Commodore 64 side of the machine, is also available in the C-128 mode. Letters in the 40-column mode are large and easy to read.

The forty-column, high-resolution graphics mode allows creation of two-color high-resolution shapes and drawings. This is a separate screen from the one used for forty-column text, and text produced with the PRINT command won't be displayed here. A special command, CHAR, can be used to obtain text in the graphics mode.

The forty-column, multicolor mode allows the use of more colors than the high-res mode, but with a sacrifice in resolution. In the multicolor mode, you can have four colors per character block (as opposed to two in high-res). Your drawings and shapes, however, will have only half the clarity.

Forty-column split text/graphics allows display of conventional text created with the PRINT command—but only below a predefined line. This text/graphics boundary is specified when the mode is first turned on.

The eighty-column text mode is well-suited for word processing and other profession-oriented software. The 80-column mode requires a more expensive RGBI monitor and the use of graphics is restricted.

Note: The high-resolution and multicolor modes are divided into a series of invisible *character blocks* (40 by 24). Each block can contain different sets of colors. Thus, if you design pictures carefully, you can incorporate a rainbow of colors into the screen, even in the high-resolution mode. Just be sure there are only two colors defined for any single character block in the high-res mode, and only four colors defined for a single block in the multicolor mode.

Fig. 2-3. The different graphic modes and monitors of the Commodore 128.

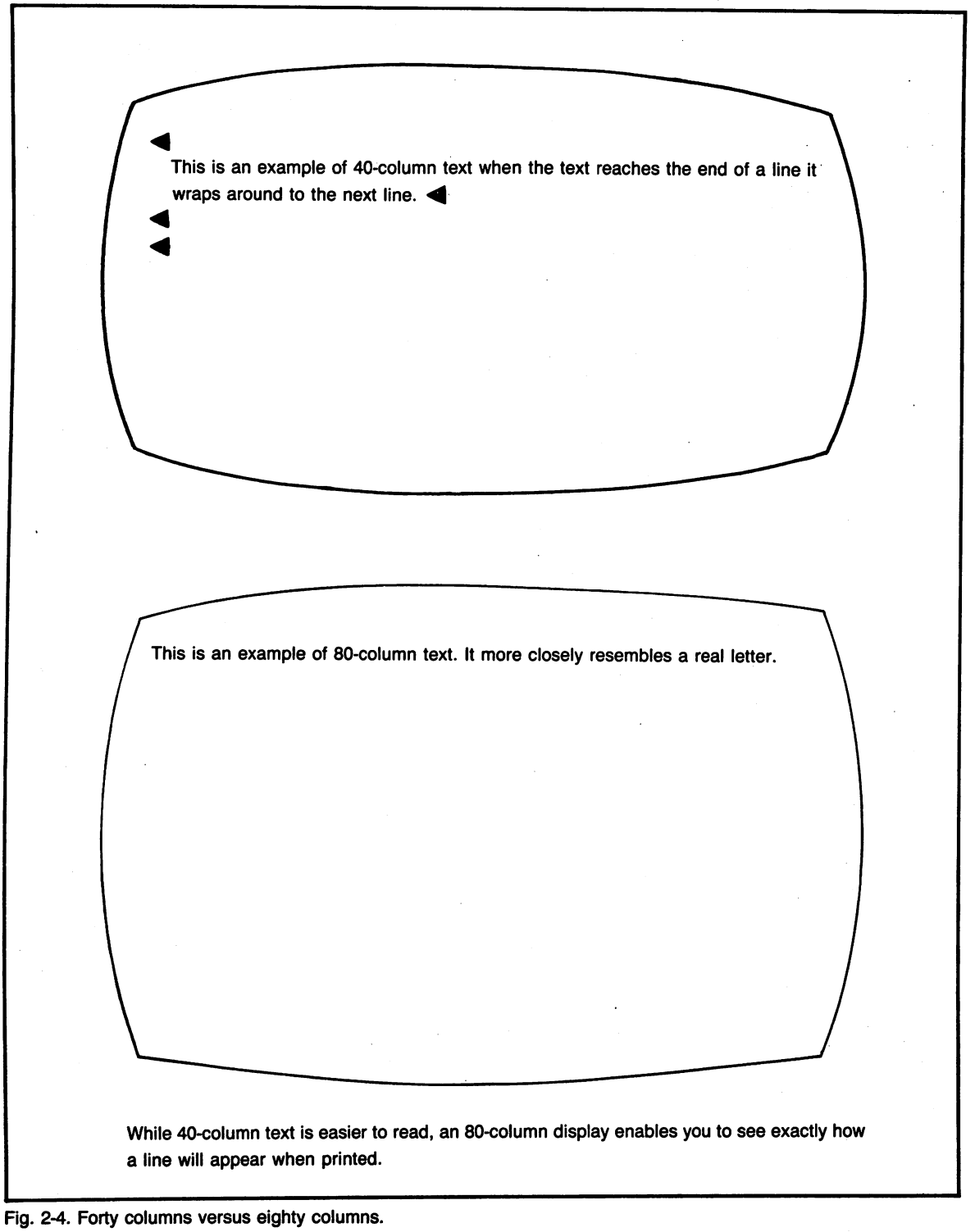


Fig. 2-4. Forty columns versus eighty columns.

you to see how it will look until the document is printed.

The eighty-column side of the machine can display its own brand of high-resolution graphics, but drawing pictures on the eighty-column side is difficult from BASIC. In fact, the screen output to eighty-columns comes from a completely different computer chip. An interesting side effect of this design is that you can watch a program running on the forty column side of the computer, and at the same time see the program listing on the eighty-column side. Information left on one screen stays there even while the other screen is operating. If you're very serious about programming, and your uncle Waldo left a big wad of bills for Christmas, you might consider buying two monitors: one with eighty columns for program editing, and the other with forty columns for watching the program operate. Those who don't have an uncle Waldo can generally get along fine with just a single monitor, or by combining an eighty-column monitor and a TV set.

Switching Between Modes

That 40/80 DISPLAY switch at the top of your keyboard (see Fig. 2-5) is a little deceptive. You'd think that pressing it would instantly switch video modes. You might assume that when the switch is up, the C-128 will display information on the forty column side; and that when the switch is depressed, all display would instantly go to the eighty-column side.

You'd be wrong on both accounts.

Yes, there is a keyboard command to switch instantly between the 40- and 80-column sides of the machines, but it has nothing whatsoever to do with the 40/80 DISPLAY button.

The 40/80 DISPLAY key determines the current graphics mode only when the machine is first switched on (*powered up* as they say in the computer manuals). So if the key is down when you turn the power on, the computer will be in the eighty-column mode. Otherwise, the system will display all text and graphics on the forty-column side.

There's a different command for switching modes once the computer is operating:

<ESC> <X>

You can use <ESC> <X> endlessly to toggle between forty- and eighty-column modes, as indeed you probably will if your system is equipped with both monitors. <ESC> <X> is also the first command to try if your computer appears to have "gone off to never-never land." A frozen or blank video display is often the result of the C-128 being accidentally thrown into the wrong graphics mode. If you don't have a monitor for the other side of the video display, you really have no way of knowing whether this has happened. Issuing an <ESC> <X> command is always worth a try (also make it standard operating procedure to check that the monitor is turned on, that the video cord is plugged into both the computer and monitor, and that the monitor power cord is plugged into the wall socket).

EDITING A PROGRAM ON THE COMMODORE 128

While your Commodore operator's guide contains some excellent tips on how to edit a program, there are a few additional tricks of which you should be aware. The first thing to keep in mind is that when you're not running a program when you're in the *edit* mode: pressing RETURN places the current line on the screen into memory. This makes the steps for changing a program line very easy:

1. List the line in question (ex. LIST 200).
2. Move the cursor using the keys shown in Fig. 2-6 and type in the change.
3. Press RETURN while the block *cursor* is still on that program line.

The first thing to keep in mind is that when you're not running a program, pressing RETURN places the line at the cursor position into memory. This makes it easy to list a program, type in changes on an existing line, and simply press RETURN to save the information. A new line will replace the old one.

It also means that if you don't like changes you've made on a given program line, you can keep the line intact simply by moving the cursor off of

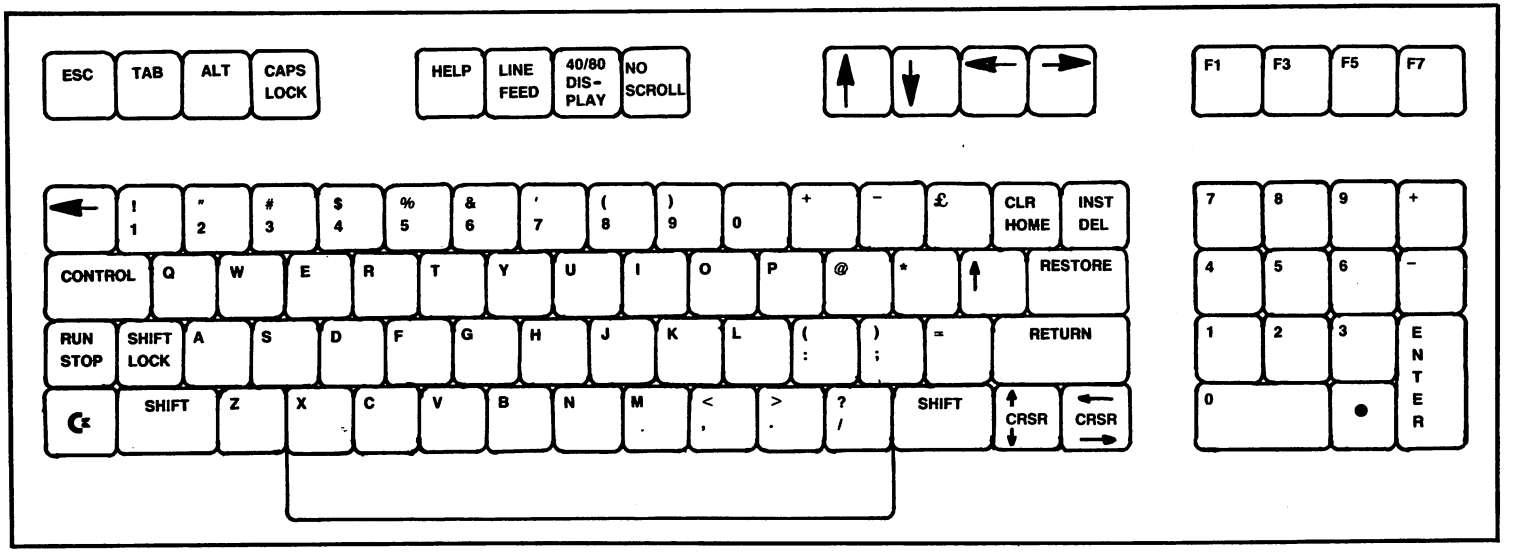
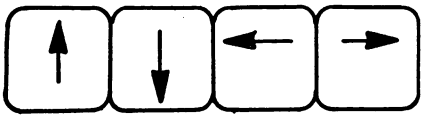
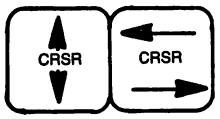


Fig. 2-5. The Commodore 128 keyboard.



The arrow keys at the top of the keyboard move the cursor in the direction indicated (at least in BASIC and in many Commodore 128 programs!). These keys do not function in the Commodore 64 mode.



The cursor keys at the bottom right-hand corner of the main keyboard are a remnant of the old Commodore 64 set. In order to move the cursor up or to the left with these keys, you must press the SHIFT key at the same time you're using the arrow key. Despite this extra effort required, many users find these lower cursor keys more conveniently placed than the arrow keys at the top.

Fig. 2-6. The two sets of arrow keys.

that line before pressing RETURN, or by clearing the screen (more about this later).

If you're not fully aware of how this "RETURN to replace" feature works, you could be in for trouble. If you just know you've made a change to a line, but later find that the change isn't there, you probably forgot to press RETURN after the edits to that line. It's a common mistake.

If you've worked with other computers that use file editors to change program lines, it's easy to forget to press RETURN to incorporate your changes. But remember: on the Commodore 128, the newly edited line isn't "in" the program until you press RETURN. It's also logical to think that the line should be in the program exactly as it appears on the screen. But you can make all the changes you want: If you don't press RETURN, the line will remain in memory exactly as it was, and no change will be installed.

Some other computers require that you *trace over* the entire edited line using the cursor keys. On an Apple, for example, you'll lose part of your line unless the cursor is sitting at the very end of it. This tracing over is not necessary on the Commodore 128, although it usually won't hurt anything.

Another rule is to carefully check the line before pressing RETURN. Once you press RETURN, everything on the current line is read into memory, even if there are stray characters on the screen at that line. Failure to check the line causes problems more often than you'd think. For instance, as shown in Fig. 2-7, if you're in certain editing modes, it's easy to wind up with the word "ready" at the end of your program lines. And it's a problem you'll usually miss until the program screams "?SYNTAX ERROR" during the middle of a run. Stray messages (such as "ready") can be easily eliminated by positioning the cursor one space to the right and pressing the DEL key a few times to back over the offending word.

Therefore check a line carefully, and eliminate any extraneous characters using the delete key. When the line appears exactly as you want it, it's safe to press RETURN.

It should be noted that when we say *line*, we mean a *program line*, even if it occupies several 40-column horizontal lines on the screen. For example, even though it stretches on for several lines, the following is considered one line for editing purposes:

```
62000 IF ANSWER="YES" THEN PRINT
      "VERY GOOD! THAT'S RIGHT":
      ELSE PRINT "SORRY THAT'S
      THE WRONG ANSWER"
```

```
1040 if a$(at)>request$ then

        high=atready.
```

Fig. 2-7. A line with READY. at the end. The word READY was picked up when this line was listed while in the ESC-A automatic input mode. The programmer apparently pressed RETURN without noticing the extra word. In many cases, this mistake will create bugs or cause the program to crash. There's one remedy: pay attention to what you're doing!

Pressing RETURN at any point (even if the cursor is at the word "wrong" or "good") would place the entire program line into memory.

Graphics Symbols and Upper- and Lowercase Letters

The Commodore (C=) Key and Keyboard Modes. The Commodore 128 offers two text *modes* that determine whether text on the screen will appear as uppercase letters and graphics symbols, or simply as uppercase and lowercase letters.

As a historical note, this dual mode for text takes its origins from the Commodore 64, and follows exactly the same rules. When the machine is turned on, it is in the uppercase and graphics mode. Unshifted letters appear in uppercase. Shifted letters appear as graphics characters. As shown in Fig. 2-8, the graphics characters produced by SHIFT are those shown on the right-front side of the key.

The lowercase and uppercase mode is the one used by word processors and other professional programs, such as the ones you'll be designing in this book.

In either mode, the number keys function just as they would on a regular typewriter; a 5 is still a 5 in either mode, and a shifted 5 produces a percent sign (%), just as you would expect.

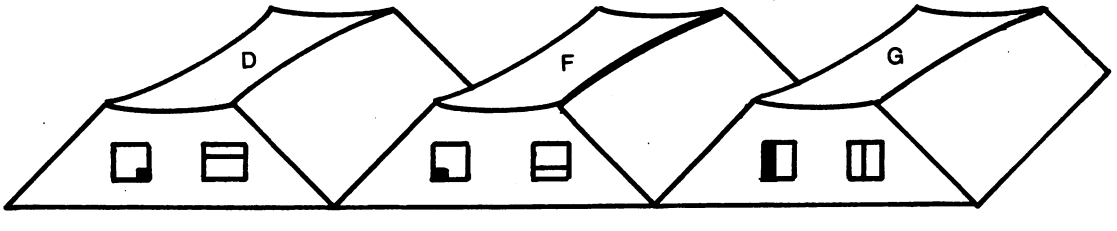
Switching from one mode to the other is easy: Simply press and hold the SHIFT key while momentarily pressing the C= (Commodore) key. All the text on the screen will instantly switch to the other mode.

One exception to all of this is the C-128's 80-column screen mode, which, as you'll see throughout this book, has its own special set of rules and privileges. In the 80-column mode, changing the text mode affects only characters that have yet to be typed; text already on the screen remains unaffected. This means shifting into the lowercase mode won't instantly turn all uppercase letters on the screen into lowercase, as it does in the 40-column screen mode. A similar set of rules holds true when you are working in any of the computers graphics modes.

Bypassing Keyboard Control. There is a way to have your computer programs switch modes automatically, with no need to press the C= key. You can even use a program command to temporarily *lock* the machine into the current mode, so users can't accidentally change into the wrong keyboard mode.

What if you want to combine lowercase, uppercase, and graphics characters all on the same screen? There are ways to do that, too. We'll explore all of these program control features in the chapter on professional appearance.

Rules for Keying in a Program. Usually, though not always, you will be typing in your program while the computer is in the upper case/graphics mode. This means the commands and statements on your program lines will appear in all



Here are three of many keyboard keys that may be used for graphics. The symbol on the left of the F key would be selected by pressing C= and F. In the upper case/graphics mode, pressing SHIFT F would produce the character on the right side of the F key.

Fig. 2-8. One of the Commodore 128's many text/graphics keys.

caps. Pressing SHIFT together with an alphabetic key will produce a graphics character in this mode.

When you're in the computers lowercase/uppercase mode, be sure your program lines appear in lowercase. If you type the program with the SHIFT-LOCK or CAPS-LOCK key down, the lines will appear in uppercase—they'll look just fine. But when you RUN the program, the C-128 will interpret all the commands as graphic symbols. The program will immediately crash.

Items enclosed in quotes are exempt from the uppercase/lowercase rule, because BASIC interprets everything in quotes literally. So this will work in the lowercase mode:

```
10 print "Hello there Sally!"
```

But this won't, because some of the commands contain uppercase letters:

```
10 rem This is a Test
20 Print Taxes
```

If you're at all confused by the different keyboard modes, try typing these two short programs in from the lowercase mode. Then RUN them. It's the easiest way to see how it all works.

Escape Commands Used for Editing

As shown in Fig. 2-9, the ESC key can be used to issue special commands that can greatly speed editing of your program.

Inserting Text. There are many times when programmers want to insert additional items of information into a program line. You might, for example, want to add an additional print statement or lengthen the name of a variable. By pressing SHIFT and INSERT/DELETE at the same time, you can insert additional spaces into a line, and then fill in these spaces with new program commands or other information. But there are two drawbacks.

First, you have to count out the number of spaces you'll need for the insert, as shown in Fig. 2-10. If you wanted to add the word PRINT to a line, you would insert five spaces for the command and one space to follow it. Counting the spaces required for long sets of statements becomes largely a matter of guess work.

The second drawback is that the cursor (arrow) keys become inactive over the inserted spaces. For example, if you made a mistake in inserting PRINT, and started the word with L instead, pressing the left arrow key would produce an inverse character and move to the right—not at all the desired result.

The Commodore 128's *automatic insert mode* is the perfect alternative to manually inserting spaces one at a time. This mode automatically inserts space for characters as they are entered, and also allows full use of the cursor keys. Especially if you're planning to do a good deal of editing on a line or groups of lines, the automatic insert mode is perfect.

The automatic insert mode is activated by simply pressing <ESC> and then the letter <A>. From then on—and until you actually run the program or GOSUB to a routine—anything typed in a line will be automatically inserted, pushing the statements following them one position to the right.

The insert mode may be canceled by pressing <ESC>and then <C> (note that these two keys should be pressed separately—it's different from using SHIFT or CONTROL in conjunction with another character).

These two commands are easy to remember:

<ESC> <A> as in "automatic insert"
<ESC> <C> as in "cancel"

Inserting Characters While a Program is Running. Because the automatic insert mode is turned off as soon as a program runs, it is not possible to use this mode to edit entry of information within an INPUT statement. But the <INSERT> and <DELETE> keys both function within a program and may be used almost anytime you're entering information.

Clearing to the End of a Line. The ESC-Q command is used to erase all characters to the right of the cursor position. For example, let's say you want to eliminate a remark at the end of a line:

The following ESC commands are useful for editing BASIC program lines. They generally do not function outside of BASIC.

LIST	Lists all program lines
LIST 200	Lists line 200
LIST 200-250	Lists lines 200-250, inclusive
ESC + A	Automatic insert mode
ESC + C	Cancels automatic insert mode and quote mode
ESC + D	Deletes the current line
ESC + I	Inserts a line at the cursor position
ESC + J	Moves cursor to beginning of line
ESC + K	Moves cursor to end of current line
ESC + P	Erases everything from beginning of the line to cursor
ESC + Q	Erases from cursor to end of line
ESC + @	Clears from cursor to end of screen window

Other ESC Commands

ESC + B	Sets bottom right corner of window
ESC + T	Sets top left corner of window
ESC + S	Changes cursor to a solid block
ESC + U	Changes cursor to an underline
ESC + E	Turns off cursor flashing
ESC + F	Makes cursor flash

Fig. 2-9. Steps for inserting text using the INS key.

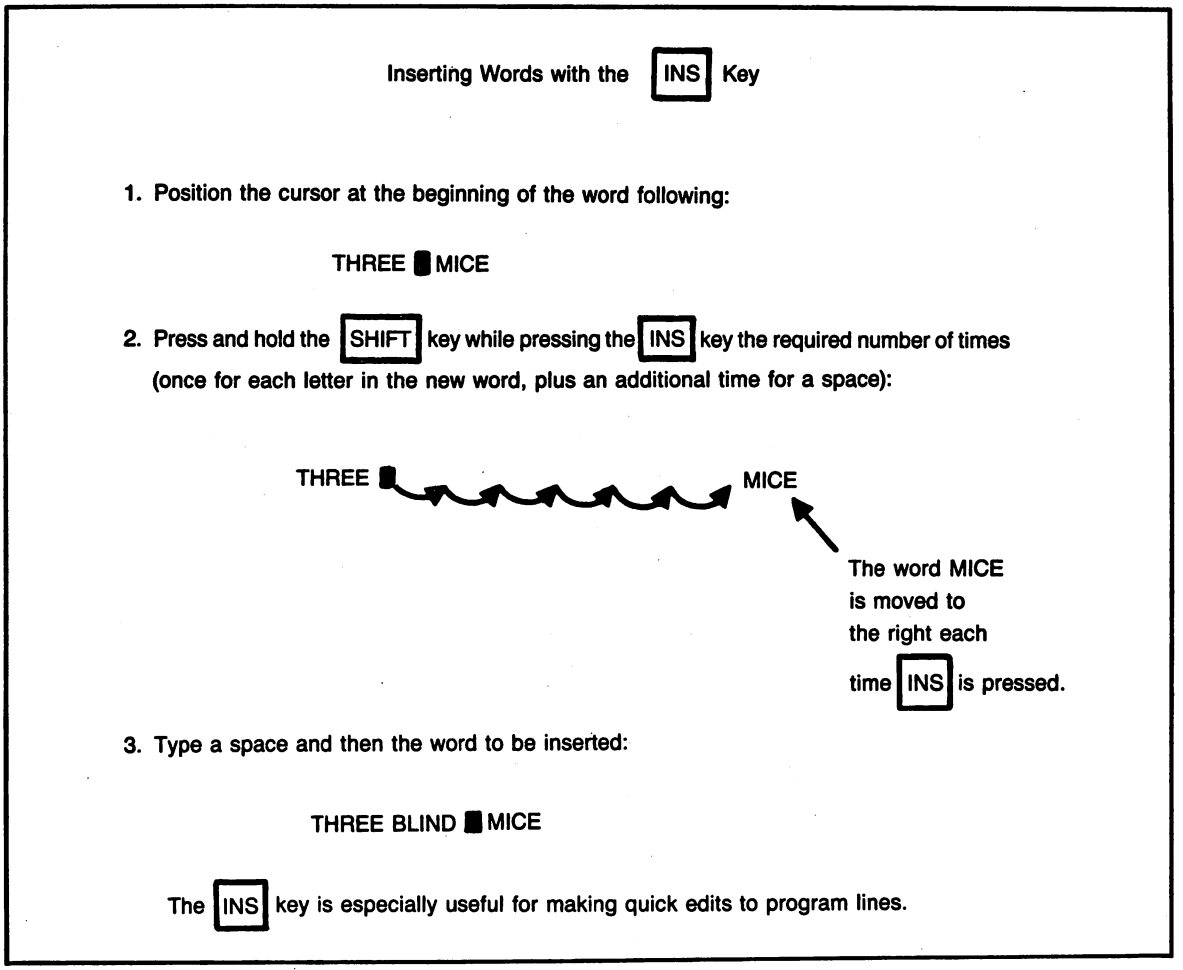


Fig. 2-10. Useful editing commands.

```
10 PRINT "HELLO"     :REM THIS LINE
                        PRINTS HELLO
```

One way would be to "space over" the remark. But this would take some time. ESC-Q provides an easier alternative. Using the arrow keys, you would simply position the cursor to the left of the colon (:), press the ESC key, and type Q. The remainder of the line would vanish. You can remember this command by simply thinking of ESC-Q as "quitting" the line at the current cursor position. Note that it's still necessary to press return for the edit to be inserted into your program.

Moving to the End of a Line. The ESC-K command is used to move the cursor to the end of a program line, and is generally used to add statements to the end on a line. ESC-K is most useful when you wish to add a statement to a long program line and you don't want to trace over 20 or 30 spaces to get to the end. Remember to type a colon (:) to separate the new program line from the old.

How to Make a Listing Pause

If you've used other BASIC's, you know that the LIST command can be used to display sections of a program on the screen:

LIST (lists all lines in a program)
LIST 200-300 (lists lines 200 through 300, inclusive)
LIST 500-980 (lists lines 200 through 980, inclusive)

When viewing a long program listing, it's often useful to be able to stop and review certain lines before they *scroll* off the top of the screen out of view. Three methods are available for doing this on the Commodore 128.

The C= (Commodore) key is used to slow down scrolling, and works as soon as the listing reaches the bottom line of your screen. Pressing C= puts the brakes on a runaway program listing, slowing things down enough so that it's possible to view every line as it goes by.

If you want to put a listing "on hold" temporarily, use the CONTROL-S command, by pressing and holding the CONTROL key while typing S. CONTROL-S stops scrolling indefinitely. When you want the listing to resume, simply type CONTROL-Q in the same way.

Finally, the RUN/STOP key may be pressed whenever you want to halt scrolling for good. To view more of the program after RUN/STOP has been pressed, you'll have to type LIST again.

Splitting and Duplicating Program Lines

You've already seen what happens when you replace statements within a line and press RETURN: The line is automatically updated. But what do you think would happen if you listed a program line and then edited its number—say, changing line number 10 to 25? Would line 10 disappear? Or would there now be two identical lines with different numbers?

If you voted for answer two, put a gold star on your calendar. Typing the number 25 over the number 10 has the effect of copying the original line. This is because the Commodore 128's line editor has no way of knowing that there ever was a line 10. It simply assumes that the line being edited should be inserted into the program at an appropriate place. Since the line number has changed, it gets inserted at a different location.

This ability to duplicate lines is a distinct advantage when you want to break up long sets of statements into two or more lines. When the duplication feature is combined with ESC-K and ESC-Q, splitting lines becomes a very simple job.

Let's say you want to split the current line into two statements:

```
10 SCNCLR: PRINT "HELLO THERE.
   HOW ARE YOU?"
```

The SCNCLR command clears the screen. The statement following it displays a simple message. Splitting these two statements onto separate lines should be relatively easy.

First, you would duplicate the line, by moving the cursor to the current line and typing a different line number (in this case, we'll use 25). After the operation, a LIST would produce this:

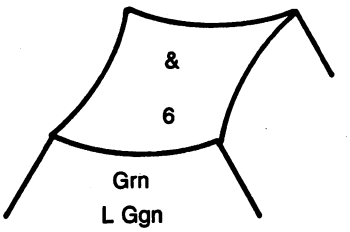
```
10 SCNCLR: PRINT "HELLO THERE.
   HOW ARE YOU?"
25 SCNCLR: PRINT "HELLO THERE.
   HOW ARE YOU?"
```

Now that you have two identical lines, you can block out the sections of both lines that are no longer needed. On line 10, you'd place the cursor at the letter R in SCNCLR, and type ESC-Q. Since everything to the right of the cursor is erased with the ESC-Q command, you'd be left with only the SCNCLR command on line 10.

On line 25, you'd do the opposite, placing the cursor at the colon after SCNCLR, and pressing the DELETE key seven times to eliminate SCNCLR. (If you're nimble enough, you could simply hold down DELETE until the word disappears, but be careful; DELETE will continue devouring characters until you release the key.) The program would now look like this:

```
10 SCNCLR
25 PRINT "HELLO THERE. HOW ARE
   YOU?"
```

From this simple exercise, you can see how



Example of a keyboard key showing colors. Green is selected by pressing CONTROL and 6. Light green is obtained by pressing C= and 6.

Fig. 2-11. A key with color.

easy it is to use the duplication and split features of the BASIC editor.

Clearing the Screen From the Keyboard

While the SCNCLR command shown in the last example is fine for use in programs, there's a far easier way to clear the screen when you're working directly from the keyboard. By pressing SHIFT and CLEAR/HOME at the same time, you can clear anything that's on the screen. This feature is especially useful when you've been performing several program operations on the screen and would like to reduce the clutter that comes with them. Pressing the CLEAR/HOME key without depressing SHIFT at the same time puts the cursor at the upper left *home* position of the screen, but does not clear the screen.

Changing Colors

One way to make program lines and other information easier to read is to change the color of characters appearing on the screen (the *foreground color*). You've probably already noticed that each of the numeric keys at the top of the keyboard is labeled with the names of two colors (see Fig. 2-11). By pressing the CONTROL key along with these numbers, the top set of colors may be selected. Pressing the C= (Commodore) key causes the bottom color on the key to be selected. These color changes affect only the color of characters and character graphics, and do not change the background color on the screen. The COLOR command, discussed in Chapter 8, is used to alter background, border, and character colors. Although you may change colors within your programs using the CONTROL-number key and C=-number key commands (within quotes), it is generally recommended that you use the COLOR command, and leave the number key color commands for use when you want to effect quick changes in color on the screen while editing. Note that the keys in the numeric keypad do not work in this manner.

Restoring the Screen

If things ever get out of hand—if you ever invoke colors that are unreadable, or the computer "hangs" for no apparent reason—you can often bail out of things by pressing the RESTORE and RUN/STOP keys at the same time. RESTORE-RUN/STOP brings the computer screen back to its original state when the power was turned on. While RESTORE-RUN/STOP clears the screen and causes many programs to halt, this special keyboard command will not erase BASIC programs from memory. (We'll talk about how to *trap* the RESTORE-RUN/STOP command in the chapter on error trapping.)

Chapter 3

A Quick Tour of DOS Commands

While the DOS Shell contained on your 1571 Test/demo disk will perform most disk preparation and maintenance functions automatically, there are a few commands and concepts it's still important to know about. They can make the difference between your being a cool, sophisticated user or a frustrated beginner.

PREPARING DISKS FOR USE

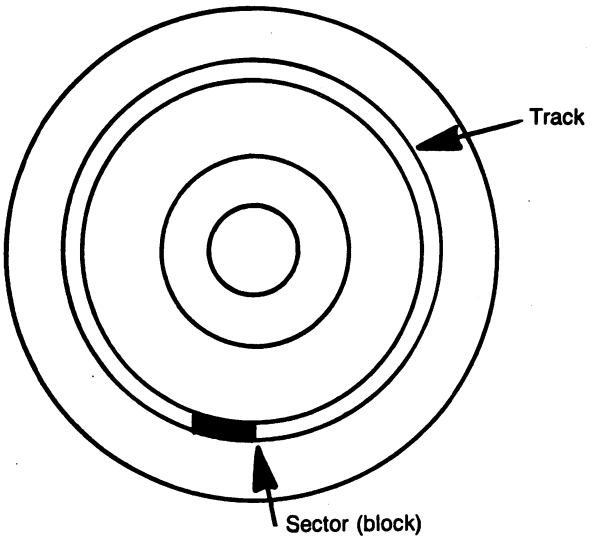
The first thing to realize is that all new disks must be specially prepared for use on your Commodore 128. If you've just taken a new disk from its box, or the disk has been used on a different type of computer (such as an Apple, IBM PC, or Radio Shack), the HEADER command must be issued before the disk can be used. HEADER erases all information that may have been stored on a disk. At the same time, the HEADER command sets up the disk for use on the Commodore 128 by creating electronic *tracks* (see Fig. 3-1) and an *allocation map* that tells the computer where everything on this disk will be stored. This procedure is known as *formatting.*

Because formatting erases everything on the disk, you should be very careful with the HEADER command. It is easily possible to wipe clean a disk containing important information. Once the HEADER process begins, you've lost everything that may have been on the disk.

The disk may be formatted either from the DOS Shell (available by inserting the 1571 disk into your drive and starting the system) or by executing the HEADER command from BASIC.

When to HEADER a Disk from BASIC

Whenever possible, format your disks from the DOS Shell. It's just as fast, and you don't have to remember all of the whys and wherefores of the HEADER command, which can get rather complicated. It's a good idea, in fact, to use the FORMAT option from the DOS shell whenever you buy a box of new disks. Simply format all 10 disks at the same



Formatted disks are made up of a series of magnetic tracks. Data is stored on *sectors* or *blocks*, within each track. A track may contain between 17 and 21 sectors depending on its position. A disk's directory is located on track 18.

Fig. 3-1. Data tracks on a magnetic disk.

time. Then store all your formatted disks in a clearly labeled box. If you follow this procedure, you'll never want for formatted disks.

There are times when using the DOS Shell for formatting is impractical. You may need to format a disk from within a program, or you may suddenly realize you're out of formatted disks.

Let's say you've just written an elegantly designed home filing program (perhaps typed from the pages of this book), and you remember that there are no formatted disks anywhere. You can't boot the DOS Shell, since doing so would wipe out the program you just typed. *This* is when you type the HEADER command directly from BASIC. If HEADER still seems a bit confusing after you've read the following paragraphs, simply format your disks by typing the HEADER command exactly as it appears in this book. As long as your blank disk is in drive zero (your first or only drive), this command will work flawlessly.

Entering the HEADER Command

HEADER is one of the trickier BASIC commands to type, because it includes four parts (see Fig. 3-2) that must always be specified to create a new disk:

```
HEADER "WORKDISK",I35,D0
```

Following the HEADER command itself is the disk title, which is always enclosed in quotes and may be up to 16 characters in length. The next item is the ID code, comprised of the letter I followed by any two-character ID code. Any letter or number is acceptable, but graphics characters, such as those obtained by using the C= (Commodore) key, won't work. Finally, a drive number must be specified: the letter D and a drive number have to be included—even if you have only one drive. If you have a single drive system, the drive designation is D0 (for drive zero). On a system with dual drives, (such as the Commodore 1572 drive unit), the designation can be either D0 or D1, depending on which drive you're using for the format.

Sometimes you'll have different drive housings, called *units*, connected to your computer. If you have two separate drives that are assigned differ-

ent unit numbers, simply include a unit number in the HEADER command:

```
HEADER "WORKDISK",I35,D0,U9
```

The command above would format a disk in drive zero, unit nine.

Usually, you'll be formatting your disks in drive zero (your first drive). Therefore, in most cases the first version of the command, without any unit number, is the one to use.

It's a good practice to perform a <SHIFT> <CLEAR/HOME> after using the HEADER command. This is an especially good idea if you're editing a program, or if you're prone to move the cursor around. There's nothing more shocking than realizing that the cursor has traced over an old HEADER command still left on the screen, and that in the process you've formatted the important disk now in the drive.

MAKING EXTRA COPIES

Another important procedure that should be performed often is the making of additional *backup* copies of your disks. The DOS Shell includes a backup procedure that works with any type of drive arrangement.

If you thumbed through the User's Guide supplied with your machine, you've probably noticed a BACKUP command. Don't get your hopes up if you have a single-drive system.

The BACKUP command only works with two-drives-in-one-unit systems, such as the Commodore 1572. For any other configuration you must use the DOS Shell to make backup copies of your disks. (It's a good idea to *boot* your system with the Demo/Test disk, so the DOS shell will always be available by pressing F1.)

Backups are more important than most people realize, especially if your disks are stored in a home environment where mishaps can easily occur.

Backups can help promote domestic tranquility. Let's look in on the home of Dan S., a Commodore 128 user who keeps all his family's taxes and other records on the computer. Dan hasn't backed up in months. He says he doesn't have time. Dan is about to learn a hard lesson involving his third-quarter family budget, a full glass of not-too-tart lemonade, and the family cat, Ralph. A quick pounce is all it takes. Dan's glass of lemonade is history. And so is his budget. Ralph has been thrown through the front window of their split level home. Dan is the talk of the neighborhood.

Susan K., another C-128 user, lives across the street. She backs up every time she updates a disk. The other day, Sue's Siberian Husky, Boris, completely devoured disks containing the first five sections of Sue's doctoral thesis on the mating habits of the East African Rock Hyrax (*Dendrohyrax brucel hindei*). "That's Okay," Sue says patting Boris

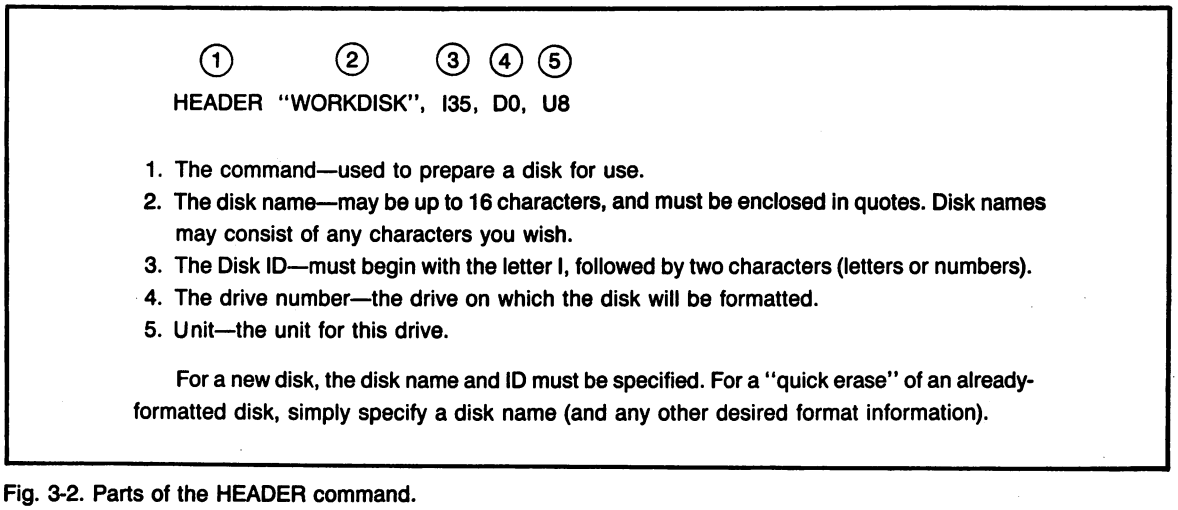


Fig. 3-2. Parts of the HEADER command.

comfortingly on the head. "I've got backups."

It will only take one unrecoverable error to convince most people how crucial backing up really is.

SEEING WHAT'S ON THE DISK: DIRECTORY OR CATALOG

Even if a disk is clearly labeled, there's nothing more reassuring than viewing a list of files right off of the disk. DIRECTORY is a command you will certainly use many times during each session with your Commodore 128. DIRECTORY, which can also be typed as CATALOG, lets you see exactly which files are on the disk and also provides specific information about file types and sizes (see Fig. 3-3). The DIRECTORY command should always be used before you save programs on a disk—both for confirming that the disk is in the drive and also for technical reasons that we will cover shortly.

Printing a Directory

There's a completely different method used for printing directories of a disk. It is not nearly as simple. In fact, this method involves loading the directory into memory—all too familiar an operation to those who once owned Commodore 64s, but one that will seem alien to most other users.

Warning: Because this procedure loads the directory into memory, it wipes out any programs currently stored in memory.

The commands for printing a directory are as follows:

```
LOAD "$",8
OPEN 1,4
CMD 1
LIST
PRINT#1
CLOSE 1
```

The why's and wherefore's of these commands are detailed later in this chapter. For now, take it on faith that when typed exactly as you see them, the commands above will list a directory on any printer that has been assigned as device 4 (the normal printer device number).

```
0 [reverse]                              [/reverse]
1     "HAPPY HOMEMAKER"   PRG
3     "SIMPLE FOOD"       PRG
4     "BETTER FOOD"       PRG
5     "DOUBLE FOOD"       PRG
3     "FILE EX 2"         PRG
3     "FILE 1 EX"         PRG
1     "FOOD FILE"         SEQ
4     "FILE EX 4"         PRG
4     "FILE EX 3"         PRG
6     "FILE EX 5"         PRG
630 BLOCKS FREE.
```

The first line (in reverse) starts with the disk name, followed by the ID number (00) and DOS version (2A).

For each file, the following information is listed: number of blocks, title and program type. One *block* represents 256 characters. A 1K program would require four blocks. A double-sided disk has 1328 blocks available, or about 340K.

Fig. 3-3. A DIRECTORY example.

Tricks with the Directory Command

If your disk is chock full of files (you can store up to 144 programs or data files on a disk) a complete directory of the disk would fill the screen several times over. The entries at the beginning quickly scroll out of view, perhaps before you've had time to get a good look at them.

Fortunately, the DIRECTORY command offers several ways to "narrow down" the names of the files you want to see. If you wanted to confirm that the file LETTER was on the disk in the default drive, you could type:

```
DIRECTORY "LETTER"
```

You could also view a directory of all files starting with the letters LE, by using a *global* feature:

```
DIRECTORY "LE*"
```

The asterisk (*) is a *wildcard* symbol that tells the computer, "Show any filenames beginning with LE, regardless of how they end." Such a command would show the following filenames, if they were on the disk:

```
LETTER
LED
LETTOR
LETS GO FISH
```

Another wildcard character, the question mark, is used to represent a single letter. The command:

```
DIRECTORY "LETT?R"
```

Would pull up only the following filenames from the list above:

```
LETTER
LETTOR
```

The wildcard features can be very useful when you are sorting through lots of disks containing lots of different programs and data files.

SAVING PROGRAMS ON DISK

Once you've written a program—indeed, periodically while you're writing it—you should store the program on disk. Until a program is stored on disk, the only copy you have is in the computer's *random access memory*, a volatile temporary storage area that forgets everything it knows as soon as the power goes out or you turn off the machine.

Frequent saving of a program ensures that a power glitch or some other electronic mishap won't wipe out your work.

Saving programs on disk is a simple procedure generally. Here's an example of the DSAVE command:

```
DSAVE "STOCK MKT SYSTEM"
```

The name of the file(program) to be saved should always be enclosed in quotes and can be up to sixteen characters in length. The filename may not start with a number, but numbers may be included after the first character (see Fig. 3-4).

There are some special rules to be aware of if you want to ensure that you've saved a program properly. If you perform the DSAVE procedure incorrectly, it's possible to think that a program has been saved when it really hasn't. It is also possible to actually damage information on a disk at the same time.

Rule #1. Always use the DIRECTORY command before saving a program. The first reason is that it's always a good idea to confirm that you're saving information on the right disk. The second reason is even more important: if you've swapped disks, the computer can sometimes become confused, thinking the old disk is still in the drive. Your program can sometimes be *written out* to areas of the disk that are already spoken for, not only garbling the program that you're trying to save, but making mince meat out of existing files on the new disk. The DIRECTORY command forces the computer to *read* the area of the disk that tells what space is available.

If you're using older Commodore drives (such as the model 4040), it's also a good idea to type:

OKAY	NOT OKAY
DSAVE "STOX PROGRAM"	DSAVE STOX PROGRAM (no quotes)
DSAVE "EXAMPLE #1"	DSAVE "1 EXAMPLE" (starts with number)

Fig. 3-4. Examples of filenames.

DCLEAR

or:

OPEN 1,8,15,"I"

in order to *clear* the drive of any information left over from the previous disk.

Once a program is saved, it will appear in the directory with a PRG file type:

8 "STOCK MKT SYSTEM" PRG

Replacing Existing Program Files

Another reason for performing a directory is that the program you plan to save may already exist on the disk. If you want to replace that file with an updated program, it is necessary to precede the program name with an at symbol (@):

DSAVE "@STOCK MKT SYSTEM"

This is a safety feature built into DOS to ensure that you don't accidentally wipe out a program by saving something different under the same name. If the program exists, and you don't include an at symbol in the command, the DSAVE will appear to have worked, but the new program won't be saved at all. The only clue the computer gives that something's wrong is a flashing drive light. The drive light won't stop blinking until the next DOS command is successfully executed. Make it a rule to check the light on your disk drive after any disk operation. It often provides the only warning that something's gone wrong.

Replacing Files on Almost-Full Disks

Commodore recommends that the @ replace feature be used only on disks that are less than half full, because the command requires a certain amount of *work space* on the disk. If you need to resave a program on an almost-full disk, erase the original file first, using the scratch command:

SCRATCH "STOCK MKT SYSTEM"

As soon as you type <Y> at the "ARE YOU SURE?" prompt, the old program file will be eliminated. You can then save the program in the normal manner (without using the replace option, since the file is no longer on the disk):

DSAVE "STOCK MKT SYSTEM"

Figure 3-5 shows the steps involved in saving a program, and Fig. 3-6 shows some common errors that can occur during saving.

LOADING A PROGRAM

Loading a program is even simpler than saving one. Simply type DLOAD followed by the name of the file enclosed in quotation marks:

DLOAD "STOCK MKT SYSTEM"

You can then type LIST to view the program or RUN to execute it. Another DOS command loads and runs the program in one operation:

RUN "STOCK MKT SYSTEM"

Keep in mind that LOAD and RUN wipe out any programs currently stored in memory. Naturally, they won't affect programs stored on a disk. These two commands will work with any BASIC program on a disk.

- Do a DIRECTORY of the disk to confirm it is the one on which you wish to save the program.
- If there is an error, make sure the disk was properly formatted. (Formatting is accomplished with the HEADER command.)
- Type the DSAVE command, the filename in quotes, and any needed drive and unit numbers.

  - Examples of saving a new program:

    ```
    DSAVE "FILENAME"
    DSAVE "FILENAME",D1
    DSAVE "FILENAME",D1,U8
    ```

  - Examples for replacing a program:

    ```
    DSAVE "@FILENAME"
    DSAVE "@FILENAME",D1
    DSAVE "@FILENAME",D1,U8
    ```

- Do a DVERIFY to confirm the file was saved correctly:

  ```
  DVERIFY "FILENAME"
  ```

Fig. 3-5. Steps for saving a program.

RUNNING OTHER FILES

Machine language or binary files are also listed in the directory as PRG file types, but because they are stored in a different format, these programs cannot be started with the RUN command. Instead, the BOOT command is used to load and execute binary programs:

```
BOOT "BINARY STOX"
```

Unless you are creating binary programs yourself, most of the software you write will be RUN. Commercially sold software almost always includes detailed instructions on how to start the program, and many of these programs start automatically when the disk is placed in drive 0 and the computer is turned on.

DISK DRIVES AND OTHER DEVICES

Just to keep things interesting, Commodore offers several different setups for disk drives, and, as shown in Fig. 3-7, each of these configurations

Problem	Solution
The disk is unformatted.	Format the disk using the HEADER command or the DOS shell.
The disk is write protected.	Remove the write-protect tab from the disk.
The name of a file is too long ("String too long").	Use a new name of 16 characters or less.

Fig. 3-6. Common problems during program saves. These problems will cause the drive light to flash or produce an error message.

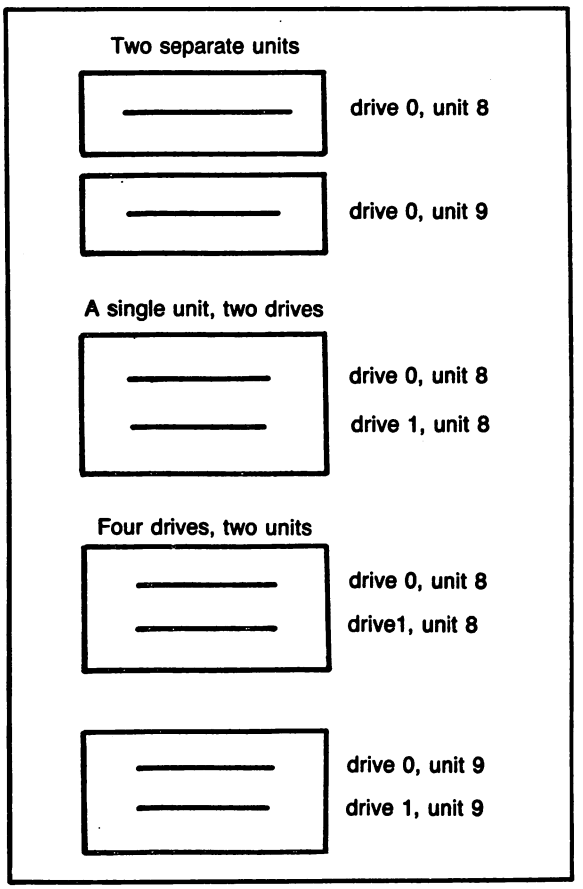


Fig. 3-7. Different drive arrangements.

gives the Commodore 128 a different way to refer to specific drives.

If you have only one disk drive and plan to add no more, then referring to individual drives is rarely a problem. The only command that will require you to specify a drive number is the HEADER command, which is used to prepare a disk, and which we'll talk about a few pages from now. The two drive numbers available are zero (for the first drive) and one (for the second drive). It gets more complicated.

If you have a dual drive system (such as the 1572) in which two drives are built into one physical housing, then the second drive is referred to a drive one. Both drives are part of *unit number* 8. Eight is the device number that Commodore long ago assigned for its first drive unit.

Two Units, Two Zeroes

Drive and unit numbers work somewhat differently with two individual drives. If you have two separate drive units, both of them will be referred to as drive zero. These drives are identified, then, not by drive number, but by unit number. The first disk drive will be drive zero, unit eight (abbreviated in commands as D0,U8), and the second drive will be referred to as drive zero, unit nine (D0,U9).

Commodore did not design the system to be confusing—it just turned out that way. But eventually, you'll get used to the idea that single disk drive units are referred to as unit eight and nine, and that dual drive setups have two drives (D0 and D1) that are part of one unit (U8, U9, U10, and so on).

Generally, the first drive connected to the system is drive 0, unit eight. Zero and one are the two drive numbers available on one dual-drive *device* or *unit*. Eight is one of several *device numbers* that drives can be assigned.

To make matters even more complex, some commands work when both drives are in one unit (such as the dual-drive 1572) but don't work if you have two separate drive units, such as 1571s or 1541s. Fortunately, Commodore has included the DOS Shell on it's Test/Demo disk; it performs many disk commands and handles all the unit/drive confusion from behind the scenes. If you're going to be performing certain disk operations, it's much easier to use the DOS Shell. For example, the DOS Shell is the only way to transfer files between separate drive units.

Accessing Devices

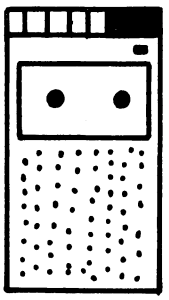
It may help to think of disk drives as simply a type of *device* that the computer controls. While the drives are referred to as unit eight or nine, other peripherals attached to the computer are referred to as devices. In fact, as far as the computer is concerned, *units* and *devices* are the same thing. For example, commands that relate to the printer generally refer to device four or five. The screen can be referred to by a device number (zero), although information automatically goes to the screen unless you direct it somewhere else. Even the keyboard

can be accessed as a device.

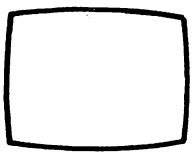
The C-128's way of communicating with devices could easily be compared with a police radio system. The transmitter is capable of sending several signals at once, but each patrol car receives only a specific signal. The Commodore 128 uses similar *channels* to communicate with different devices. Commands being sent to one device go only to that device. If the printer's channel is selected by a program, all output goes to the printer. Other devices that are not "tuned in"—such as disk drives or *modems*—ignore the signals being sent over the serial cable. Figure 3-8 is a chart of different devices and channels with examples of how to assign each one.

SENDING INFORMATION TO DIFFERENT DEVICES

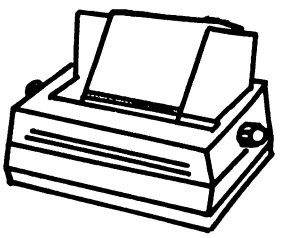
The OPEN and CMD commands are used to transfer *output* to devices other than the screen.



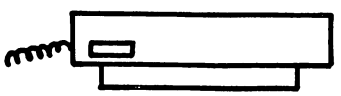
Cassette Recorder	to read:	OPEN 1,1,"FILENAME",0:CMD1
	to write:	OPEN 1,1,"FILENAME",2:CMD1



Screen	OPEN 1,3:CMD1



Printer	OPEN 1,4:CMD1 OPEN 1,5:CMD1 OPEN 1,6:CMD1 OPEN 1,7:CMD1



Modem	OPEN 1,2

By specifying the proper device number in the OPEN command, you can access many devices. To redirect screen output to one of these devices, use CMD 1 (assuming you assigned a file number of one). To print a single line to one of these devices, use the PRINT # command. For input, where appropriate, use INPUT#.

Fig. 3-8. Devices and how they are opened.

1. "OPEN" the file number and device. Example:

```
OPEN 1,4
```

2. Redirect the output. Example:

```
CMD 1
```

3. Display information as usual. It will be redirected to the new device.
4. Clear the channel using PRINT#.

```
PRINT #1
```

5. Close the file:

```
CLOSE 1
```

Fig. 3-9. Steps to make a device active.

CMD is most often used to print information that normally would be displayed on the screen. Here are the steps needed to print the word "HELLO" on a Commodore printer that is assigned device 4 (the usual number):

OPEN 1,4	(Open a line of communication to the printer)
CMD 1	(Transfer all output to channel 1)
PRINT "HELLO"	(Print the word)
PRINT#1	(Clear out channel 1 to printer)
CLOSE 1	(Close channel 1)

The first command opens up a channel (think of it as an electronic file) to which it assigns the printer (device four). While the channel (file) number in this example is 1, it could be any number from 0 to 255. Your manual also refers to this channel number as a *file*, and in fact it is a sort of an electronic file—directed only at the printer.

The CMD 1 command on the second line instructs the computer to turn over all output to file (channel) one, which we previously defined as the printer. From here on out, all text that would normally go to the screen will appear instead on the printer.

The PRINT "HELLO" statement prints the word HELLO on paper. Note that this command will not be printed itself. CMD 1 redirects only screen output to the printer; the CMD 1 command does not affect information that is typed from the keyboard.

The final two commands on the list are used to empty out the printer file. The PRINT# and CLOSE statement should always be used to turn off any device that has been activated by OPEN and CMD. Always remember to clear the file with PRINT#, and then to CLOSE it. Otherwise it will remain open and may cause errors when you attempt to perform other printer or disk-related operations.

Another way to do the same thing would be to include a *write string* after CMD; this eliminates the following PRINT "HELLO" line entirely. Here's what it would look like:

```
CMD 1,"HELLO"
```

As you can remember, there are two commands (OPEN 1,4 and CMD 1) that redirect output from

the screen to the printer. And there are two more that turn off or close out the printer. You can use this type of structure any time you wish to print information. Just place as many print statements as necessary between the two sets of commands. If you want to print some lines on the screen and some on the printer, simply turn the printer on and off using the OPEN, CMD, PRINT#, and CLOSE commands.

Using Duplicate File and Device Numbers

Many users like to use the same file and device numbers, so they can easily remember which file (channel) is assigned to what device. Using this approach, the commands would look like this:

```
OPEN 4,4
CMD 4
PRINT "HELLO"
PRINT#4
CLOSE 4
```

Note that these commands can be included in a program or typed directly from the keyboard.

Sending Files to Disk Drives

The OPEN feature is used in a slightly different way when a program must send information into a data file. The sections of this book on Sequential Files and Random Files cover the Commodore 128's abilities in this area extensively.

Sending Files to the Screen

As odd as it seems, there are some occasions when you might want to treat the screen as a file. For example, what if you wanted to let the user decide whether information should be displayed on the screen or sent to a printer. If you simply predefine file 1 based on the user's response, the CMD 1 command can be used in your program to direct output to the right place:

```
10 INPUT "TO SCREEN OR PRINTER (S/P)";ANS$
15 :
20 IF ANS$="P" THEN OPEN 1,4: ELSE OPEN 1,3
25 :
30 CMD 1
```

1. Make sure the disk has been prepared (if the disk is new, use the HEADER command, or format it using the DOS shell).
2. Type DIRECTORY to view the disk's contents.
3. Type the DSAVE command:

   If saving for the first time:

   ```
   DSAVE "MY PROGRAM"
   ```

   If saving to replace:

   ```
   DSAVE "@MY PROGRAM"
   ```

4. Check the drive light to make sure that it is not flashing after the READY message appears.
5. Type the DIRECTORY command to confirm that the file was placed on the disk.
6. Perform a DVERIFY to ensure that the program was saved correctly:

   ```
   DVERIFY"MY PROGRAM"
   ```

Fig. 3-10. Steps for saving a program.

```
35 :
38 :
40 FOR X=1 TO 10
50 : PRINT "THIS IS A TEST"
60 NEXT
65 :
68 :
70 PRINT#1
80 CLOSE 1
```

The key to this routine is line 20, which chooses which channel to open, based on the value of ANS$. Once the file has been defined, the program sends information to the appropriate pipeline. The CMD simply redirects output to this file, whatever it may be.

While all of this may seem complicated right now, by the time you've completed this book it will have become second-nature to you. Bear with the strange disk drive designations, device numbers, and files, and you will become a proficient and respected Commodore 128 programmer. Figure 3-9 will serve to remind you how to open and send information to devices other than the screen.

Chapter 4

Searching for Information

Applications for searching are wide ranging indeed. They involve much more than the simple location of a name or an address. For example, you can use searches to have your programs automatically look up and process information. In addition to performing calculations and storing information, one of your computer's principal functions is to search for information.

HOW TO SEARCH

While there's no "search" command that will find information, there are many methods of searching for data with a computer.

Essentially, all searching methods involve finding a match-up between two or more items. In fact, you're probably familiar with the simplest type of search, even if you've never written a search routine before.

Take a look at the HAPPY HOMEMAKER program, listed in Fig. 4-1, which contains as simple a search as you're likely to find anywhere. After accepting entries from the husband and wife, the program looks for a match between the word YES and the contents of the HUSBND$ and WFE$ variables. If either of these variables is equal to the word YES, line 90 instructs the computer to print "Good, I'll make some then."

The computer has in effect looked for a match within two variables (HUSBND$ and WFE$) for the word YES, taking a different action depending on whether or not it is found.

If you're a little unsure of how the program works, try typing it in and running it. A few trial runs should make clear what's happening internally as you give different responses to the two questions.

The procedure in the Happy Homemaker program works splendidly with two guests. But what about a list of five guests? Or ten? The same procedure could be repeated in an expanded form, but look at how complicated things become with five variables being used:

```
90 IF A$="YES" OR B$="YES" OR
   C$="YES" OR D$="YES" OR
   E$="YES" THEN PRINT "GOOD, I'LL
   MAKE SOME"
```

```
10 rem :perfect hostess program:
20 :
30 print"would either of you like more coffee?"
40 :
50 input "husband's response";husbnd$
60 :
70 input "wife's response";wfe$
80 :
90 if husbnd$="yes" or wfe$="yes" then print "i'll make some then"
95 :
110 end
```

Fig. 4-1. A happy homemaker program.

Certainly a much easier way would be to store the responses in a list, or *array* of items, which could be scanned in a subroutine. Fortunately, the Commodore 128 has a special way of handling lists, one which makes them very easy to search for information.

DINNER AT THE COMMODORE INN

Think about five food items. Think then about stacking them in a list, such as a luncheon menu. Then think about how you would find out if hot dogs were available.

"Well," you say, "I would look for the words *'hot dog'* "

But if you think for a moment, you'd really do much more than that. You'd actually look at *each item* on the menu, and—in split second—mentally decide whether it was "hot dog."

If you wanted to search the list in the computer for a particular set of words or characters (such as HOT DOG), wouldn't it be simple to have the computer test for a match between each individual item and the words HOT DOG. If the computer discovers a match, it could print the word "found" or even go down to a separate subroutine to perform whatever actions you desire. Conversely, if no match is found the computer could perform a different set of actions or simply end the program.

While immensely flexible, such a search routine actually requires very few program lines to execute. Our program will have three subroutines:

1. A procedure that asks users what to search for (what they want for lunch).
2. A routine to place a food list into memory (see Fig. 4-2).
3. A routine to search through the food list and determine if the requested item is available.

Data Entry

The job of the first routine, shown in Fig. 4-3, is simply to find out what the customer wants. There's nothing pretentious about it:

Putting the Data into an Array

Placing the information into memory can be done in many different ways, but for this example we'll take the most direct approach, as shown in Fig. 4-4. By placing a RETURN statement at 60100, this becomes a subroutine that can be called from any section of our test program.

Tricks with the Array

One of BASIC's strongest features is its ability to refer to information through the use of *variables*. This holds true for string arrays, such as FOOD$, which can be referred to not only by constant numbers (1, 2, 3, and so on) but also by variables (J, K, BL, and so on). Thus, in our array, the following statements would both print the same word ("Hamburger"):

Example 1. PRINT FOOD$(2)
Example 2. AT=2: PRINT FOOD$(AT)

Note that Example 2 is using a variable within a variable, to refer to the second item in the FOOD$ array.

In our search routine, shown in Fig. 4-5, the variable AT will be used to point to the item number that is currently being examined for a match. This routine performs a DO . . . UNTIL loop, checks each item in the FOOD$ list for a possible match against the REQUEST$ string (REQUEST$ will hold the name of what the user is searching for on the Commodore Inn's Bill of Fare). If the item

A PAPER LIST	A COMPUTER LIST
1. Chili Dog	ITEM$(1) = "Chili Dog"
2. Hamburger	ITEM$(2) = "Hamburger"
3. Pizza	ITEM$(3) = "Pizza"
4. Paella	ITEM$(4) = "Paella"
5. Filet Mignon	ITEM$(5) = "Filet Mignon"

Fig. 4-2. Placing lists in memory.

```
5000 rem    :input routine:
5020 print "what would you like"
5040 input "for dinner ";request$
5060 :
5070 :
```

Fig. 4-3. Entering a request.

Fig. 4-4. Loading data into an array.

```
60000 rem     :load up list:
60010 food$(1)="chili dog"
60020 food$(2)="hamburger"
60040 food$(3)="pizza"
60060 food$(4)="paella"
60080 food$(5)="filet mignon"
60090 last=5:   rem  there are 5 items
60100 return
```

```
1000 rem    :search routine:
1005 found=0:at=0
1010 do until at=last
1020 :   at=at+1
1040 :   if request$=food$(at) then print"yes, we have that!":exit
1060 loop
1080 return
1090 :
```

Fig. 4-5. Searching for data in BASIC.

is found, the program prints "Yes, we have that!" exits the loop, and then returns from the subroutine.

A few lines at the top are needed to run the program, which is shown as a whole in Fig. 4-6.

Line 10 *dimensions* the variable FOOD$, which is another way of saying that the program is reserving in memory for this list of information. While it's not required that you dimension an array that will have fewer than ten items, using the DIM statement is a good practice to get into, because most of your lists will contain at least ten items.

Line 20 clears the 40-column screen using the SCNCLR 0 command. (Zero is the 40-column screen number. There are many different display screens on the Commodore 128, and many different ways to clear them, all of which will be discussed thoroughly in the chapters on graphics and professional program design.)

Lines 30, 40, and 50 call the subroutines indicated in each REMark statement to the right. Line 70 ends the program.

Again, all the lines containing solitary colons (:) are included simply for appearance; they divide up the lines, but don't affect program operations.

Even though you're searching through a longer list of information than in the first example (where

```
5 rem   :simple food:
10 dim food$(5)
20 scnclr 0
30 gosub 5000 :        rem get entry
40 gosub 60000:    rem list into memory
50 gosub 1000 :    rem do search
60 :
70 end
80 :
1000 rem    :search routine:
1005 found=0:at=0
1010 do until at=last
1020 :   at=at+1
1040 :   if request$=food$(at) then print"yes, we have that!":exit
1060 loop
1080 return
1090 :
2000 :
5000 rem    :input routine:
5020 print "what would you like"
5040 input "for dinner ";request$
5060 :
5070 return
60000 rem     :load up list:
60010 food$(1)="chili dog"
60020 food$(2)="hamburger"
60040 food$(3)="pizza"
60060 food$(4)="paella"
60080 food$(5)="filet mignon"
60090 last=5:   rem  there are 5 items
60100 return
```

Fig. 4-6. A complete listing of the Simple Food program.

there was only HUSBND$ and WFE$ to contend with), you can see that the logic remains simple. The heart of any search routine involves matching pairs of items. This same search routine could be used to scan a list of ten items, or a list of one-hundred.

Computer searches always involve telling the computer to look for a match between two items.

IMPROVING THE SEARCH ROUTINE

Of course there are many things we can do with our search routine now that it's installed. The first thing you'll probably notice is that the program does little more than indicate whether or not the item is available. And even this it does by cramming several statements on a single line following the test for a match (line 1040). While the approach works well enough in small programs, it's quite difficult to read. And if you ever plan to expand the program, this type of structure can become quite unwieldly.

A solution to these problems is the use of what computer programmers call a *flag*. A flag is defined as any variable used to mark whether or not a particular event has occurred. In this example, you could choose the variable name FOUND to indicate whether or not a particular match occurred (whether a particular food item was found). You could use the FOUND flag anywhere in the program to determine whether the customer had selected a food item that is on the bill of fare.

Figure 4-7 shows a slightly modified listing with line 1040 getting FOUND equal to −1 when the search is successful (it will become clear in a few pages why we've use the unusual value of negative one).

Notice that we've moved the "Yes, we have that" message out of the search subroutine and into its own special routine shown below. If FOUND equals −1, the routine prints the same message as before. If FOUND is not equal to −1, the program prints the message "Sorry, we're all out today."

```
510 IF FOUND=-1 THEN PRINT "YES,
    WE HAVE THAT":ELSE PRINT
    "SORRY, WE'RE ALL OUT TODAY"
```

The ELSE statement is useful in a variety of circumstances, but as you become more familiar with BASIC 7.0, you'll probably find yourself using it primarily to make decisions on line numbers or math operations:

```
IF FOUND=-1 THEN GOSUB 10000 :
        ELSE GOSUB 15000
IF FOUND=-1 THEN X=20 : ELSE X=10
```

Using ELSE to print message tends to clutter up your program, but ELSE works very well in applications such as these.

While using ELSE may seem like overkill in line 510 above, there will be many times when you wish to perform groups of operations based on whether or not an item exists in a list. For example, if a requested item didn't exist in an address list program, the computer could test the flag and branch to a routine that asks if the user wants to add the name to the current list. Right away, the

```
1000 rem    :search routine:
1005 found=0:at=1
1010 do until at=last
1020 :   at=at+1
1040 :   if request$=food$(at) then found=-1:exit
1060 loop
1080 return
1090 :
```

Fig. 4-7. Search routine using a FOUND flag.

use of a variable flag has given us greater flexibility, because the program "knows"—at any point after the search—whether the requested item has been found, and it remembers throughout the rest of the program.

Look what else our FOUND flag has bought us! A DO . . . UNTIL loop that works on FOUND and lets the program continue requesting luncheon orders until a food item is located. Not only has the flag neatened up the program and made it more readable; it has also allowed us to be more flexible with what we do outside of the search subroutine.

The Easiest Kind of Test: True/False

There's another trick we can use to make the program even more readable. A kind of programming shorthand that enables you to remove the = −1 part of the test is available when you are using flags.

Here's an example:

```
IF FOUND=-1 THEN PRINT "FOUND
YOUR ITEM!"
```

and

```
IF FOUND THEN PRINT "FOUND YOUR
ITEM!"
```

perform exactly the same function. But the second statement is more readable, requires fewer keystrokes, and actually executes on the Commodore 128 a split-second faster than the first statement.

The same thing can be done with a DO . . . WHILE or DO . . . UNTIL loop:

```
DO UNTIL FOUND=-1
```

becomes:

```
DO UNTIL FOUND
```

There's only one small hitch: this trick works on the Commodore 128 only when you use a flag set at (equal to) negative one (−1). If FOUND = 0, for example, the program would still consider the item NOT FOUND. Setting the FOUND variable equal to other numbers such as +1, +5, of −3 yields confusing results. Now you know why we used −1 to set the flag in our previous example.

Variables that are used in this manner are called *Boolean variables* or simply *Booleans*, after the English logician George Boole, who figured out more than a hundred years ago that mathematical variables can be used to label things as TRUE or FALSE.

To the Commodore 128, a variable equal to −1 is always considered TRUE (when used in the type of test above), and a variable equal to zero in this type of test is always considered FALSE.

Accentuating the Negative

There's one other trick we can perform with Booleans—one that again will make your program easier to read. Let's say that instead of using a DO . . . UNTIL loop as in the example above, you decide to use a DO . . . WHILE structure. You've probably already figured out that DO . . . WHILE and DO . . . UNTIL are opposite ways of accomplishing the same thing. In our first example:

```
DO UNTIL FOUND=-1
```

and

```
DO WHILE FOUND=0
```

would have exactly the same effect (because when FOUND is set to −1, it can no longer equal zero, and the loop will end). It probably won't surprise you to know that there's a shortcut to this negative way of looking at things, too. As we mentioned previously, to the Commodore 128 a flag equal to zero is considered NOT TRUE (in other words, FALSE). If this is the way a test for TRUE works:

```
DO UNTIL FOUND
```

how do you think a test for NOT TRUE would work?

To test for NOT TRUE (FALSE) flag setting, simply place NOT in front of the variable name:

```
DO WHILE NOT FOUND
```

is the same as:

```
DO WHILE FOUND=0
```

which is the same as:

```
DO UNTIL NOT FOUND
```

As we use these Boolean representations throughout this book, you'll see how much cleaner and more "English-like" they make the programs appear. Generally, this book will not use the negative approach shown above, since its operation can become confusing on the C-128. We will incorporate the Boolean FOUND test into our next search example, which illustrates how related bits of information may be displayed after a search.

SEARCHING FOR RELATED INFORMATION

So far we've been using routines that pluck a given item from a list based on a positive match. There are many times, however, when you'll want your programs to do much more than to simply verify whether a particular item exists in a list. For example, in an address listing, you might want the computer to display an individual's address and telephone number, in addition to simply confirming that the person is on the list.

Let's go back to our Commodore Inn example, where we find that the chef has added a new feature to the bill of fare. Now, in addition to the main course (such that it is), a customer gets two side dishes. For instance, with a hamburger you now get french fries and a salad. With a chili dog, you get soup and creamed corn.

These types of additional items are easy to represent on the computer. You'll remember that our last example used a single list of five items stacked in a single column. What if we added two more columns to represent the two side dishes? We'd then have three columns of information, five rows deep, as shown in Fig. 4-8.

Information arranged in this way is called a *two-dimensional* array. The first dimension (row), represents one of the five luncheon selections available at the Commodore Inn. The second dimension (column) is used to refer to either the main course (column 1) or the two side dishes (columns 2 and 3).

This type of arrangement is quite useful when you want the computer to store and retrieve sets of related information.

In our last two trips to the Commodore Inn, we used statements such as:

```
FOOD$(2)="HAMBURGER"
```

to refer items in the list. We also substituted a variable for the numeric constant to come up with:

```
IF REQUEST$=FOOD$(AT) THEN . . .
```

This trick allowed the computer to refer to different items by simply changing the value of AT. But how do we handle rows of items that have more

ROW	COLUMN 1	COLUMN 2	COLUMN 3
1.	Chili Dog	Soup	Creamed Corn
2.	Hamburger	French Fries	Salad
3.	Pizza	Moon Pie	Salad
4.	Paella	Plantains	Black Beans
5.	Filet Mignon	Rice Pilaf	Salad

Fig. 4-8. A three-column menu array.

```
60000 rem      :load up list:
60010 food$(1,1)="chili dog":food$(1,2)="soup":food$(1,3)="creamed
      corn"
60020 food$(2,1)="hamburger":food$(2,2)="french fries":food$(2,3)=
      "salad"
60040 food$(3,1)="pizza":food$(3,2)="moon pie":food$(3,3)="salad"
60060 food$(4,1)="paella":food$(4,2)="plantains":food$(4,3)=
      "black beans"
60080 food$(5,1)="filet mignon":food$(5,2)="rice pilaf":food$(5,3)=
      "salad"
60090 last=5:    rem  there are 5 items
60100 return
```

Fig. 4-9. Loading a two-dimensional array in BASIC.

than one column? The solution is to add another number within the parentheses:

FOOD$(2,1) = "HAMBURGER" :
FOOD$(2,2) = "FRIES" :
FOOD$(2,3) = "SALAD"

Here, the first number (in parenthesis) refers to the row (all items are in row two). The second number refers to the column. As you can see, the main course is listed under row one, and the two side dishes are listed as being under rows two and three. Arrays provide a very fast and efficient way of handling rows and columns of information on a computer.

Now that we're using a two-dimensional array, our load-up-list routine will change somewhat, as shown in Fig. 4-9.

We'll have to change a few other lines that previously referred to a single array and now must contend with a double (two dimensional) one. The first line for editing is 10, which must now be changed to a double-dimension:

10 DIM FOOD$(5,3)

When you're working with arrays, it's important to remember that a variable cannot be turned into single and double dimensions within the same program. For example, the statement in line 20 below would give an error, because A$ had already been defined as an array:

10 DIM A$(10) :REM GOOD STATEMENT
20 DIM A$(10,10) :REM BAD STATEMENT (Second dim!)

There's nothing wrong with the line 20 statement itself. It's just that the array is already "spoken for," since a single-dimension array has already been defined at line 10.

If we changed the second variable to B$, everything would be OK again, because B$ has not yet been dimensioned. You could also have a separate nondimensioned B$ (a B$ with no parentheses) variable, which would cause no conflicts. As you work with BASIC, you'll find the only conflicts that arise are between different types of string arrays using the same variable, as in the A$ example above.

The next line to change is 1040, where we actually test for the item being searched. In this case, the variable

FOOD$(AT)

has been changed to

FOOD$(AT,1)

which represents row number AT (the ATth row) in the first column. You'll remember from discussions on the preceding pages that one is the column that contains the main course.

SEARCHING: WHAT THE COMPUTER KNOWS

One of the first rules of this type of programming is to figure out what your computer "knows" and take full advantage of it.

Let's review what the computer "knows" in this program:

1. It knows how many items there are, based on the value of LAST.
2 It knows what the customer has ordered (what's being searched for) through the REQUEST$ variable.
3 It knows the names of all items and side dishes on the menu (through FOOD$(AT,1), FOOD $(AT,2), etc.).
4. If an item is found, it knows by the variable AT exactly which row the FOOD$ item is on.

Using these simple bits of information, an enormously powerful program can be constructed. Naturally, the computer doesn't really understand the significance of LAST or AT, but we've designed the program so that it makes decisions based on the values of these variables.

Now that the search and data loading sections are complete, we have only to decide what to do with the information once it has been found. Since The Commodore Inn's waiters are known for their politeness, they'll mention the side dishes that accompany the meal, just so the customer will know what's coming.

Exactly what do we want to print? First, a confirming message that the item does exits:

```
PRINT "THAT'S VERY GOOD TODAY"
```

Next, we'll want to display the name of the item our customer has selected along with its two side dishes. you already know that AT in this program is used to mark the row number where the item was found. You also know that column one always contains the name of the item selected, and that columns two and three always contain the side dishes. Think about what this means: you can refer to all three items in a certain row by using the variable AT as a *pointer* to that row. Tying all of this together produces the following:

```
PRINT "WITH THE "FOOD$(AT,1)" YOU GET"
PRINT FOOD$(AT,2)" AND "FOOD$(AT,3)"."
```

The first line prints the item selected. The second prints the names of your side dishes. The display would look like this:

```
WITH THE PAELLA YOU GET
PLANTAINS AND BLACK BEANS.
```

Of course, the message will change depending on what item was selected.

Coping with Extra-Long IFs

Naturally, this new set of statements calls for some remodeling of the PRINT MESSAGE ROUTINE at lines 500-600.

As you've probably already gathered, the list of what's available is more than a simple IF statement can handle: There's just too much information to fit gracefully into one line. If you tried, it would look something like this:

```
IF FOUND THEN PRINT "WITH THE "FOOD$(AT,1)" YOU GET":PRINT FOOD$(AT,2)" AND "FOOD$(AT,3)"."
```

Granted, in this case, the line would execute well enough. But it's unsightly as it stands, and if you wanted to add any more statements you'd be stuck; there's not much more room.

The classical solution would be to print all responses for FOUND in a separate subroutine:

```
510 IF FOUND THEN GOSUB 700 :REM DO REQUEST$ FOUND MESSAGES
```

With all statements tucked neatly out of sight at line 700, there's no question that this approach works. Yet it adds yet another subroutine to track through. Perhaps a better solution is one that Commodore has added to its BASIC: the ability to perform statements based on a test, even if the test is several lines above. The IF . . . BEGIN in Fig. 4-10 lets us add three program lines that will only be executed if FOUND is true. The BEND (pronounced B—END) at line 518 tells the program that this special if logic is ended. After a BEND, the computer continues executing each statement in the normal manner.

You can see how this type of program, shown in its entirety in Fig. 4-11, is just a short step away from one that searches through address lists or a daily schedule.

SEARCHING FOR PARTIAL ITEMS

The program examples we've used so far have one restriction: they only work with exact matches. So if you requested a HAMBURG or BURGER, the waiter wouldn't know what you were talking about. The program would come up empty and ask for another selection.

The solution, of course, is the Commodore's INSTR function, which was briefly reviewed in Chapter 1. In that example, we used INSTR to verify that the input variable Y$ contained certain characters (Y or y or N or n):

DO WHILE INSTR("YyNn",Y$)=0

In effect we were asking the computer to search the first string (a literal enclosed in quotes), for an occurrence of the second string (Y$). If there's a match anywhere in the first string, the INSTR function will be equal to the position of the character, as shown in Fig. 4-12. For example, if Y$ contained the letter <N>, this statement would return the value three.

INSTR is a lightening-fast and immensely valuable function that is easily included in search routines, such as the one in our Commodore Inn programs.

The format of INSTR is always the same:

X-INSTR (string1,string2, position)

String1 is the string to search through. String2 is the string to search for. Position represents the character place in string1 where the search should begin (INSTR always scans from left to right). As we'll see in the section on data storage, this position features has real advantages when you're scanning for a letter or group of letters that may occur more than once in a string. For now, be content that the positioning function works if you need it—you won't be using it in the next few examples.

If no starting position is specified, INSTR automatically starts the search at the beginning. We'll be using this default setting in the following examples.

Translating string1 and string2 into literal strings (characters enclosed in quotes), shows how

```
500 rem    :print message routine:
510 if found then begin
512 :  print"that's very good today!"
514 :  print"with the "food$(at,1)" you get"
516 :  print food$(at,2)" and "food$(at,3)"."
518 bend
520 if not found then print "sorry, we're all out today"
530 :
540 return
550 :
```

Fig. 4-10. Basing the response on a flag.

```
5 rem     :double food:
10 dim food$(5,3)
20 scnclr 0
25 do until found
30 :  gosub 5000 : rem get entry
40 :  gosub 60000: rem list into memory
50 :  gosub 1000 : rem do search
55 :  gosub 500  : rem print message
60 loop
65 :
70 end
80 :
500 rem    :print message routine:
510 if found then begin
512 :  print"that's very good today!"
514 :  print"with the "food$(at,1)" you get"
516 :  print food$(at,2)" and "food$(at,3)"."
518 bend
520 if not found then print "sorry, we're all out today"
530 :
540 return
550 :
1000 rem    :search routine:
1005 found=0:at=0
1010 do until at=last
1020 :   at=at+1
1040 :   if request$=food$(at,1) then found=-1:exit
1060 loop
1080 return
1090 :
2000 :
5000 rem    :input routine:
5020 print "what would you like"
5040 input "for dinner ";request$
5060 :
5070 return
60000 rem     :load up list:
60010 food$(1,1)="chili dog":food$(1,2)="soup":food$(1,3)="creamed
      corn"
60020 food$(2,1)="hamburger":food$(2,2)="french fries":food$(2,3)=
      "salad"
60040 food$(3,1)="pizza":food$(3,2)="moon pie":food$(3,3)="salad"
60060 food$(4,1)="paella":food$(4,2)="plantains":food$(4,3)="black beans"
60080 food$(5,1)="filet mignon":food$(5,2)="rice pilaf":food$(5,3)=
      "salad"
60090 last=5:   rem  there are 5 items
60100 return
```

Fig. 4-11. A complete example of a program using two-dimensional arrays.

The Commodore 128's INSTR command may be used anytime you wish to look for characters. The most common applications are searching and input screening routines. The command's format, INSTR(A$,B$), translates as "search the contents of A$ for the group of characters contained in B$." Naturally, either of the strings may be a literal group of characters enclosed in quotes, such as "Hello" or "YyNn". If no match is found, INSTR(A$,B$) equals 0. If there is a match, this function returns the position where the match was found:

Value Returned	Y$	
INSTR("YyNn",Y$)=1	"Y"	(Y is the 1st position)
INSTR("YyNn",Y$)=2	"y"	(y is the 2nd position)
INSTR("YyNn",Y$)=3	"N"	(N is the 3rd position)
INSTR("YyNn",Y$)=4	"n"	(n is the 4th position)
INSTR("YyNn",Y$)=0	no match	(Anything other than what's contained in the first item)

This particular INSTR test would be a fast and efficient way of checking the user's answer to a question that should only have a yes or no answer. See the text for a program example.

Fig. 4-12. Examples of INSTR.

INSTR might be of use in the Commodore Inn program:

```
X=INSTR("HAMBURGER", "BURGER")
```

This example is a little unrealistic, because INSTR usually works with string variables (A$, B$, FOOD$(AT,1), etc.), instead of literals. After all, you can just *look* at these two strings and know there's a partial match. But you can see immediately the applications to which this command can be put. In the above case, INSTR would find the word BURGER within HAMBURGER, and make X equal to 4. In this case, we only care that X>0 (item was found) or X=0 (item wasn't found), but other applications of INSTR make good use of the value returned by this function.

If we were to substitute variables (FOOD$ for what's on the menu and REQUEST$ for the item requested), the appearance formula would change, but the result would be the same.

```
X=INSTR(FOOD$,REQUEST$)
```

Take a moment now to look at the DOUBLE FOOD program in Fig. 4-11. At what line would you install an INSTR search? And how would you handle the fact that FOOD$ in this program is a two-dimensional array?

Both answers come in one neat package: You would replace the current match-up test (line 1040) with this new line:

```
1040 IF INSTR$(FOOD$(AT,1) ,RE-
     QUEST$ ) > 0 THEN FOUND=
     -1:EXIT
```

Pop this line into the DOUBLE FOOD program, and you'll be able to type HAMBURG or BURGER for a hamburger and DOG for a chili dog.

If you're afraid that the customer might enter "I WOULD LIKE A HAMBURGER PLEASE", you could reverse the test—this time looking for FOOD$(AT,1) within REQUEST$. By adding another line to the program you could even check for both matches:

```
1040 IF INSTR$(FOOD$(AT,1)
     ,REQUEST$ ) > 0 THEN
     FOUND= -1:EXIT
1042 IF INSTR$(REQUEST$ ,FOOD$
     (AT,1) ) > 0 THEN FOUND=
     -1:EXIT
```

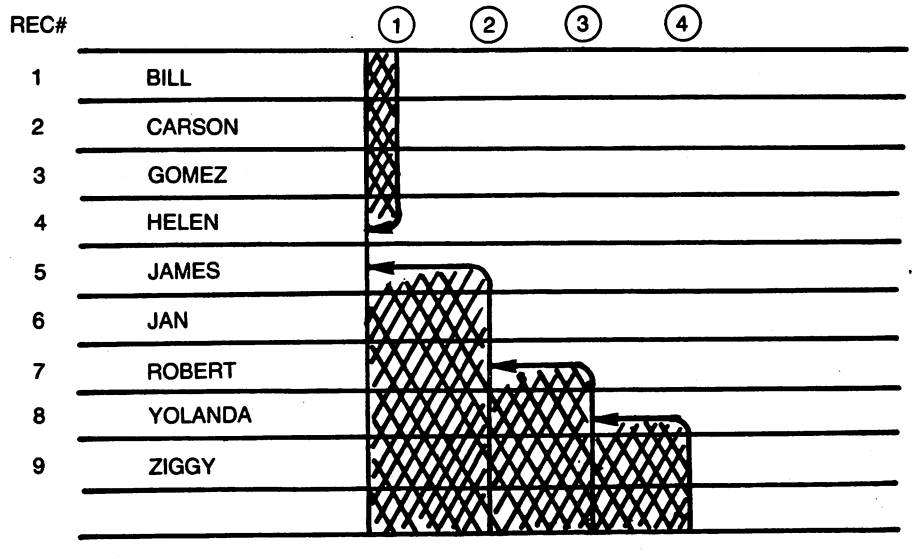
which translates as: The item is found if the customer enters part of the name, or if the customer request is found as a partial match, following a scan of the food item list.

TOWARD FASTER SEARCHES

The type of searches you've seen so far are fairly quick—as long as you're only dealing with lists of a few dozen items. The real world is a little different. What about lists of a few hundred items? These could clearly take a lot longer. Because the search routines we've been dealing with so far check every item, you can readily see how a search of a big list can be quite time consuming.

Of course, long before computers came along folks were grappling with the problem created by long lists. They found the answer long before Charles Babbage invented the first analytical engine and the computer age was born.

Think for a moment about the time-tested ways of storing and looking up information on paper: dictionaries, encylopedias, address directories, and later, telephone books. To find information in any one of these, you can simply open to the middle and start looking. If your looking for Pete's Plumbing Supply and the first entry you run across is Carson's Canary Emporium, you can instantly eliminate all the pages on the left side of the open book; they won't contain what you're looking for. Next, you take the inch or so of pages that's left, and split it, figuring that one of the halves will contain Pete's. This time you open to Stella's Sandwich Shoppe—and you instantly know that Pete's is the first half of the pages. This mission of search and eliminate would continue a few more times until you hit Pete's Plumbing Supply right on the button. There are always a few misses this way, but it's a lot more



A diagram of a binary search. Search areas (shown by shading) are reduced with each successive test. A binary search assumes elements in the file or array have been sorted.

Fig. 4-13. Looking for Yolanda: A binary search.

```
0 :          goto 9
2 : binary search
3 :
4 : low is always lower limit
5 : at is always test location
6 :
7 :
8 :
9 request$="xx"
10 do while request$<>""
11 scnclr
12 gosub 200 :rem request
14 gosub 60000
15 found=0:at=0
20 gosub 1000
30 if found then print "found":else print "not found"
35 getkey a$
40 loop
199 :
200 input "request ";request$
210 return
299 :
1000 low=0:high=10
1010 do until found or (high-low=1)
1020 at=(int((low+high)/2)+.5)
1030 print at
1040 if a$(at)>request$ then high=at
1050 if a$(at)<request$ then low=at
1060 if a$(at)=request$ then found=1
1070 loop
1090 return
1099 :
60000 a$(1)="bill"
60010 a$(2)="carson"
60020 a$(3)="gomez"
60030 a$(4)="helen"
60040 a$(5)="james"
60050 a$(6)="jan"
60060 a$(7)="robert"
60070 a$(8)="yolanda"
60080 a$(9)="ziggy"
60090 return
```

Fig. 4-14. Programming a binary search.

practical than going through the entire phone directory, name by name, to find Pete's.

The key to this system, of course, is that all the names in the phone book are stored in alphabetical order. Provided you save your computerized information in the same way, the Commodore 128 will have no trouble performing the same trick.

Programmers refer to this type of procedure as a *binary search* because it divides the list into progressively smaller and smaller parts. Figure 4-13, "Looking for Yolanda," illustrates how a list would be divided up from start to finish, with the program splitting the list first at JAMES, then at ROBERT, and finally locating Yolanda as item number eight.

Figure 4-14 translates this design into a search routine. If you're dealing with big lists, this routine could easily replace the previous examples we've used at lines 1000-2000. And here's the magic: binary searches still require only a few samples of the data (say, 10 tests) when doing 1,000 items.

Searching for Partial Matches

Because it relies so heavily on alphabetic arrangement, binary searches cannot be used to find one string that is part of another; the INSTR function used in our last example simply won't work. If you're willing to add a few lines to your routine, however, you can design a program that will search for items based on the first few characters. This type of search enables you to find JAMES by typing JA, ROBERT by typing R, and so on. The first step is to figure the number of characters the user has typed; this is available through BASIC's LEN function:

```
5060  LR = LEN(REQUEST$)
```

You can then locate the record by matching the first characters of each item in the array with the characters being searched for:

```
1035  TEMP$= LEFT$(A$(AT),LR) :REM
      LOOK FOR SHORTER A$(AT)
1040  IF TEMP$ > REQUEST$ THEN
      HIGH=AT
1050  IF TEMP$ < REQUEST$ THEN
      LOW=AT
1060  IF TEMP$ = REQUEST$ THEN
      FOUND = -1:EXIT
```

Line 1035 simply lops off part of the array string so it can be properly matched with the item requested. If the user types JA as a search request, TEMP$ will reflect a two-character version of each item in the A$ array (BI, CA, GO, HE . . .). The remainder of the routine works in the same way as before.

Of course, it's always possible that several items in the list begin with the same letter (JAMES, JAN), so its essential that a routine be included to "backscan" for other possible matches:

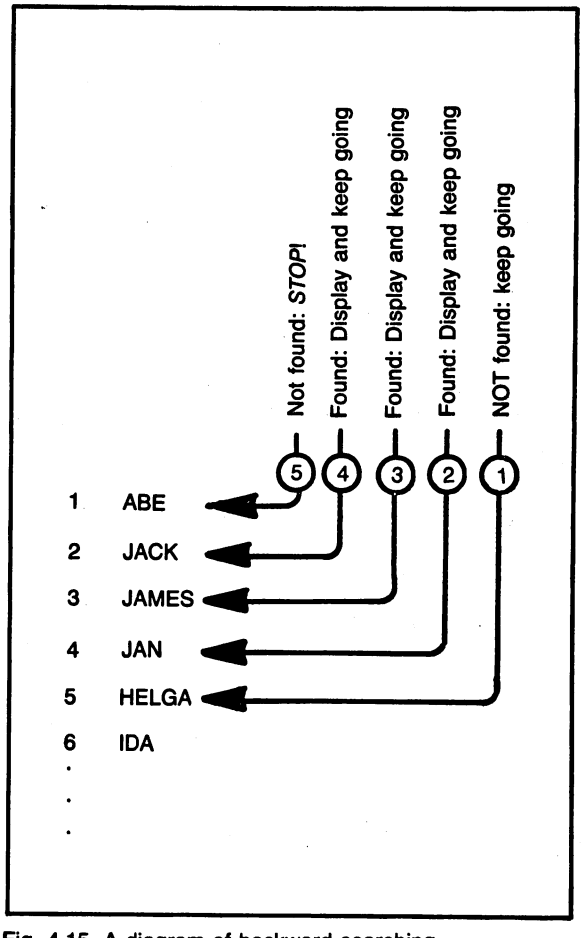


Fig. 4-15. A diagram of backward searching.

```
1200 REM :BACKSCAN ROUTINE:
1210 DO UNTIL (LEFT$(A$(AT),LR)
     < > REQUEST$) AND AT < > 0
1220 AT=AT-1
1230 LOOP
1240 AT=AT+1 :REM RESET AFTER
     LAST TEST
1250 RETURN
```

The test is done at the very top of this DO/WHILE loop, so the only other statement necessary decrements AT with each pass, thus scanning the list backward for possible matches. Because the list is in alphabetical order, it's assumed that AT is pointing to the first occurrence of JA once no match is found. The program proceeds to line 1240 which simply pushes AT to the value it held before the last test. When the routine concludes, AT is equal to the first occurrence of JA (or whatever characters are being searched for). This procedure is illustrated in Fig. 4-15.

This type of routine is best used in conjunction with another one that allows the user to view successive records in the list. After all, it wouldn't be much use if all the user could see were JAN, whether he searched for JAN, JAMES, or JACK!

Difficulties with Binary Searches

While they're extremely fast, binary searches do present a few programming difficulties. First, to use them, all your data must be kept sorted at all times. We'll be dealing with different sorting and indexing methods in Chapter 8.

Another drawback is that if you wanted to locate records by several different criteria (last name, address, city), you'd have to resort the file each time a different type of search were called for (you can't locate a city with a binary search if the list is stored by last name). While an *index* on each type of item can be kept, this amount of data handling begins to get complicated. Thus binary searches are not for everyone. If you plan to create files containing large amounts of information—and you want to access that data quickly—then a binary search is just fine for you. But you may want to get your feet wet with a simpler type of searching routine, and later replace that routine in your program with a binary search.

Chapter 5

Storing Your Data

There's nothing more satisfying than creating and using your own files. When you can successfully store and recall your own data on disk, you have clearly left the ranks of the beginners. You're starting to extract some real performance from your system.

Although most computer owners get the jitters when it comes to writing their own filing programs, or even simple file routines, file handling is not the quagmire of complex actions that many make it out to be. In fact, once you understand the concept, you'll wonder why your programs haven't made fuller use of files in the past. Those who choose not to use files are missing out on one of the Commodore 128's more savory ingredients.

HOW DISK FILES WORK

Sending information into a disk file is much like printing it on a screen. The only difference is that printing information into files must be done especially carefully; when the program goes to retrieve file information, it must know exactly how that information was stored originally.

Take a look at Fig. 5-1 to see how the bill-of-fare information used earlier could be printed on the screen. This listing would display all of the information in the bill-of-fare, assuming this data has previously been placed in memory. The listing itself would be unformatted—just a simple printout of information that would begin like this:

```
CHILI DOG
SOUP
CREAMED CORN
HAMBURGER
FRENCH FRIES
SALAD
. . .
```

This is exactly the way in which information is stored in a file. It may at first appear confusing, because there are no column headings or other keys to the information. But if the program is written properly, the computer will recall this information in exactly the right order every time. If you wanted

```
100 gosub 60000
200 gosub 6000
300 end
6000 rem    :list info:
6010 for row=1 to 5
6020 :  for col=1 to 3
6030 :    print food$(row,col)
6040 :  next col
6050 next row
6060 return
60000 rem     :load up list:
60010 food$(1,1)="chili dog":food$(1,2)="soup":food$(1,3)="creamed
      corn"
60020 food$(2,1)="hamburger":food$(2,2)="french fries":food$(2,3)=
      "salad"
60040 food$(3,1)="pizza":food$(3,2)="moon pie":food$(3,3)="salad"
60060 food$(4,1)="paella":food$(4,2)="plantains":food$(4,3)="black
      beans"
60080 food$(5,1)="filet mignon":food$(5,2)="rice pilaf":food$(5,3)=
      "salad"
60090 last=5:   rem  there are 5 items
60100 return
```

Fig. 5-1. The bill of fare listing.

to type in this program, just load the DOUBLE FOOD program converted earlier and delete the body of the program, leaving only the 60,000's:

DELETE 0-59999

Then type in the program listing in Fig. 5-1. This approach will save you the effort of entering all of the food items again.

OPENING A DATA FILE

Now that you've seen what information looks like when it's stored, let's see how data is actually written into a file. The process is almost as easy as sending it to a screen. The difference is that each file must be assigned a name and number, so that the Commodore 128 will know how to refer to it. This is quite similar to the concept of *defining* the printer as a file, covered in Chapter 2. The DOPEN command is used to open up a file on disk. In the first example, we will use DOPEN to create a new file:

6005 DOPEN #1,"FOOD FILE",W

This line instructs the Commodore to open file on disk under the name FOOD FILE. We've chosen to call this file number one within the program, although we could pick any number up to 255. Throughout the program, when it's necessary for statements to refer to this file, it will be called simply #1—there won't be any other references to FOOD FILE, because your file has already been defined for the computer. Once a file has been opened, the C-128 prefers to deal with file numbers, not names, so #1 will always be used in our example.

The ,W at the end of this line tells DOS that this file is being opened so that it may be *written* to. If you wished to *read* (recall) information from the disk, no such additional labels would be necessary, because DOS generally assumes you'll want to read a title unless told otherwise. Figure 5-2 shows some examples of the DOPEN command.

So far, we've opened up a file, but very little action has taken place. The name of the game when using disk files is storing and recalling information, and that's the next step in our program.

STORING DATA IN A FILE

A few pages back we said that sending data into a file is quite similar to printing it on the screen. As we proceed, you'll notice that all of the commands used for data handling are strikingly similar to those employed for screen display and keyboard input. The command to write, or print, information into file number one is:

```
PRINT#1
```

PRINT#1 can be used with variables, literals (items in quotes), or a combination of the two:

```
PRINT#1,A$
PRINT#1,"HELLO THERE"
PRINT#1,A$+"HELLO THERE"
```

Here, BASIC file numbering system becomes essential, because it not only tells DOS that you want to output information to a file: it designates which file. Remember, once a file is open, BASIC always refers to the file as a number. In a program that uses several files simultaneously, you can print data into alternate files as easily as a conductor points to members of an orchestra:

```
6030 PRINT#1,"Here are this year's Belmont
     Stakes jockeys"
6035 PRINT#2,"Here are the horses for the
     Belmont this year"
```

The headings above would each be placed in

DOPEN #1,"@TAX INFORMATION",W	opens up a file named TAX INFORMATION. This file will be *written to* (information will be saved into it). This file will be referred to as #1. The at symbol (@) tells DOS that existing information in the file should be replaced.
DOPEN #1,"TAX INFORMATION"	opens up the TAX INFORMATION for *reading* (loading) of information (there's no ,W at the end).
DOPEN #5,"GAME SCORES"	opens up a file named GAME SCORES for reading as file number five (note that there DO NOT have to be four previous file numbers selected; a file number may be any you choose).

Because it performs several operations at once, the workings of DOPEN can sometimes be difficult to grasp. It may help to think of DOPEN as doing all the things that need being done before your program can read (recall) or write (store) information to a file:

1. DOPEN assigns a file number, which you choose. This number is used to refer to file operations as long as the file remains open. You may choose any number from 1 to 255.
2. DOPEN names the file. The name can be any you choose, up to sixteen characters long. It may not begin with a number. If you use a literal name (TAX INFO), the name must be enclosed in quotes. If the name is a variable, such as A$, it must be enclosed in parentheses.
3. The DOPEN command tells the computer whether the file will be used for writing or reading. A file may be open for reading or writing, but never both at the same time. If you want to read form a file that is open for writing, or vice versa, simply close the file in the current mode, and reopen it in the new mode.

Fig. 5-2. Examples of the DOPEN command.

```
60000 rem      :load up list:
60010 food$(1,1)="chili dog":food$(1,2)="soup":food$(1,3)="creamed
      corn"
60020 food$(2,1)="hamburger":food$(2,2)="french fries":food$(2,3)=
      "salad"
60040 food$(3,1)="pizza":food$(3,2)="moon pie":food$(3,3)="salad"
60060 food$(4,1)="paella":food$(4,2)="plantains":food$(4,3)="black
      beans"
60080 food$(5,1)="filet mignon":food$(5,2)="rice pilaf":food$(5,3)=
      "salad"
60090 last=5:   rem  there are 5 items
60100 return
```

Fig. 5-3. Data in a two-dimensional array.

two distinctly separate files (presumably one file for jockeys and the other for horses). If the program writer desired, there could be additional PRINT# statements to send information to still other files.

You've probably already guessed that the PRINT# command may also be used with arrays, such as the one you've encountered many times before: the bill-of-fare at the Commodore Inn. Figure 5-3 shows that by substituting PRINT#1 for PRINT in the screen display loop a few pages earlier, you can easily convert this routine from displaying information on the screen to writing it into a file.

Note that the only line that has changed is 6030, which prints the array into a file instead of onto the screen. Two new lines have been added: 6005 (which opens the file for writing as #1), and 6060 (which CLOSES the file).

The DOPEN command in 6005 has something new: an at symbol (@), which instructs DOS to replace this file if it already exists. It's the same system used when you wish to replace a BASIC program that already exists on the disk. The @ symbol is important to include in programs that will write over an existing file. If this symbol is not present when you try to replace a file, the drive light will begin flashing, an error will occur, and none of the new information will be placed into the file.

The DCLOSE statement in 6060 is just as important as a DOPEN, because it tells the computer you're done with this file for the moment.

When you are writing information onto a file, as in this example, DCLOSE is vital, because it forces the computer to cough up any stray bits of information that were PRINTed to the disk file but that may not have reached the disk itself. Closing a file is necessary because the Commodore 128 (and most other computers) send file data to a temporary holding *buffer* before the information is actually written to the file. When enough data is stored in the buffer, DOS dumps everything from the buffer onto the disk, exactly in the order it was written. Closing simply forces the Commodore 128 to send any remaining data out to disk. The computer considers work on this file complete for the time being. Once a file has been closed in this manner, it cannot be referred to again by the program until it is reopened.

Many programmers like to add a general close-all-files statement at the exit to their programs, by including DCLOSE without any numeric reference:

```
DCLOSE
```

This ensures any file accidentally left open during program operation will be closed upon exiting. Generally, it's a good practice to close a file as soon as your file reading or writing routine is done with it. There's nothing worse than losing data because the power goes out and data is still sitting in a file buffer.

```
7000 rem    :read from file:
7005 dopen #1,"food file"
7010 for row=1 to 5
7020 :  for col=1 to 3
7030 :    input#1,food$(row,col)
7040 :  next col
7050 next row
7060 dclose #1
7070 return
```

Fig. 5-4. Reading data from a file.

READING INFORMATION FROM A FILE

Generally, a routine that reads a file will appear almost identical to the routine that created and wrote the file in the first place. Figure 5-4 shows the routine which would be used to recall the data stored in our FOOD FILE. Only the statements on lines 7005 and 7030 have changed. The rest of the program remains essentially the same. First, since the file is being opened for reading, the open command must change:

```
7005 DOPEN #1,"FOOD FILE"
```

The command to open a file for reading is simpler than the one used for writing.

Because the file will be read (instead of being written to) the ,W is no longer appropriate; remember, the default in DOPEN is for reading. We've also eliminated the @ replace symbol, since it isn't needed in a read operation.

The other change is to the line that PRINTs data into the file in our previous example. This routing now *reads* information. An INPUT command replaces PRINT:

```
7010 INPUT#1,FOOD$(ROW,COL)
```

The statement INPUT#1 follows the same format as PRINT#1—the number sign and the word INPUT must not be separated.

Operationally, INPUT#1 is similar to BASIC's keyboard-oriented INPUT statement, except that INPUT#1 retrieves information from a file. Like its cousin, INPUT#1 presents some drawbacks in more demanding program situations. In a few moments we'll talk more about these difficulties and what you can do to steer clear of them in your disk input routines.

Setting Your Rules and Sticking to Them

In our example, there are 15 items—five rows of three columns each. The program must be structured in a way that reads all of these items—and reads them in the correct order. For example, if the data were stored in columns instead of by row (yielding records of three rows by five columns), the file would be markedly different from the example we've been working with so far. (See Fig. 5-5 for an example of how this would work.) All five main courses would be listed together, followed by ten side dishes. If the file information were read back in any other order (a row by column order, for example), the result would be very confusing. Thus, it's important to read and write your data in exactly the same order each time.

A Table of Data Items

#	*ITEM*	*COST*	*VENDOR*
1	pencils	5.23	RABB's
2	pens	7.15	DAN's
3	clips	2.81	station's

Listed by column within row	**Listed by row within column**
pencils	pencils
5.23	pens
RABB's	clips
pens	5.23
7.15	7.15
DAN's	2.81
clips	RABB's
2.81	DAN's
stations	stations

Fig. 5-5. Storing data by row and column or column and row.

TELLING YOUR PROGRAM WHAT'S IN A FILE

You'll remember we mentioned that one of the difficulties in working with files is that you cannot see what's going into them. Another difficulty is that you don't always know what items are contained in the file. When your program is reading information, this becomes of critical importance, because the computer must know how many items of information it is reading.

If the file contains five sets of items and the computer reads six, the last group in the array of best will be blank. At worst, there will be an error that will bring your program to an abrupt halt.

If the file contains five sets of data and the computer is only instructed to read four, you will have lost one of your records. It will still be on the disk, but the data from the final record won't have been read into memory. In programs that use files, it is very important to know how many records there are. There are several ways to do it:

Solution 1: Build the number of records into the program.
Solution 2: Store the number of records in the file itself.
Solution 3: Add an end-of-file marker.

Each of these solutions works, but some are better in certain circumstances.

Reading a file is like driving down lonely road without a map. You've got to carefully pay attention to the landmarks or you'll never get to your destination.

"Tell me," says the stranger as he peers into the eyes of the gas station attendant. "How do you get to the Commodore Inn?"

"Well, it's like this. You go down seven miles—it'll be on your left."

"Noooo!" The man who has been quietly rocking in front of the ice machine sits up and takes notice. "No, that's not it! Don't tell him ta go that way. Look, jest take her down til ya see the end-of-road marker. That'll be the Commodore Inn there on yer left."

Actually, both the old timers are right. They're simply giving different directions for the same thing. Reading to the end of files works in the same way.

Solution 1: Place the Number of Records Right in the Program. One solution is to "hard wire" into your program the number of items to be stored. We've done this in our previous examples by specifying 5 as the end of the FOR . . . NEXT loop. This approach works well enough as long as the number of records doesn't change. But usually you'll have a fluid number of items in your files. And with them method, unless you modify the program each time an item is added or deleted, there's no way for the program to know, from one reading to the next, exactly how many records exist.

Solution 2: Store the Number of Records in the File Itself. One of the most common approaches is to write the number of records right at the beginning of the file. The program can read this number first, and instantly know how many records to click off in the FOR . . . NEXT loop:

```
7007 INPUT#1,LAST    :REM GET # OF
     LAST RECORD
7010 FOR ROW=1 TO LAST :REM
     SLIGHTLY MODIFIED FOR . . .
     NEXT
```

As long as the program has written the number of records into the file, this number can be used as a *pointer* to the last item in the file. You will note that the number of records (LAST) must be placed at the very beginning of both file routines. Otherwise, your program will become confused and every item will be read into the wrong slot in the array. Remember that the file read and write routines must handle data in exactly the same order.

Figure 5-6 is a complete program listing that incorporates all the file handling tricks we've covered so far.

Lines 100-400 make up the Main Control Routine that *calls* the available subroutines. First, the initial data is placed into the array using the old routine at line 60000. Next, this information is writ-

```
5 rem       :file ex 3:
100 gosub 60000 :rem data into memory
110 :
200 gosub 6000: rem write to file
210 :
300 gosub 7000: rem read from file
400 end
6000 rem    :write to file:
6005 dopen #1,"@food file",w
6010 for row=1 to 5
6020 :  for col=1 to 3
6030 :    print#1,food$(row,col)
6040 :  next col
6050 next row
6060 dclose #1
6070 :
7000 rem    :read from file:
7005 dopen #1,"food file"
7010 for row=1 to 5
7020 :  for col=1 to 3
7030 :    input#1,food$(row,col)
7040 :  next col
7050 next row
7060 dclose #1
7070 return
60000 rem     :load up list:
60010 food$(1,1)="chili dog":food$(1,2)="soup":food$(1,3)="creamed
      corn"
60020 food$(2,1)="hamburger":food$(2,2)="french fries":food$(2,3)=
      "salad"
60040 food$(3,1)="pizza":food$(3,2)="moon pie":food$(3,3)="salad"
60060 food$(4,1)="paella":food$(4,2)="plantains":food$(4,3)="black
      beans"
60080 food$(5,1)="filet mignon":food$(5,2)="rice pilaf":food$(5,3)=
      "salad"
60090 last=5:   rem  there are 5 items
60100 return
```

Fig. 5-6. A simple filing program.

ten into the file.

Finally, the file is recalled and its contents read into memory.

In summary, there are two rules to keep in the back of your mind when you're using files:

1. Know exactly what you're writing to the disk
2. Read what you've written in exactly the same manner

Ninety five percent of your debugging time with files will involve the application of these two simple guidelines. The remaining five percent of your problems will probably have something to do

1. Open the file.
2. Write the desired information into the file. Be sure to include the number of records.
3. Be sure to DCLOSE the file to update the disk directory.

Fig. 5-7. Steps for writing a file.

with failure to close the files you have opened (especially when you're writing information). Figure 5-7 shows the steps for writing a file, and Fig. 5-8 shows the steps for reading a file.

Solution 3: Using an End of File Mark. Another method of reading files involves simply placing a marker at the end of the file, to signal the file's end. An appropriate end-of-the-road mark might be three dollar signs, or some seldom-used graphics character. When the computer encounters this marker, the program can be set to stop reading the file. Naturally, this calls for some type of internal counter to keep track of how many records have been read. A typical subroutine would look like this:

```
7000 REM            :READ FILE ROUTINE
7005 ROW=1:REM :INITIALIZE ROW
7010 DO WHILE A$(ROW,COL) < > "$$$"
7020   :FOR COL=1 TO 3
7030   : INPUT#1,A$(ROW,COL)
7035   : IF A$(ROW,COL)="$$$" THEN
         EXIT
7040   :NEXT COL
7015   :ROW=ROW+1 :REM :BUMPED
        UP ONLY IF NOT END
7050 LOOP
7060 RETURN :REM :END OF ROUTINE
```

Note that because the $$$ code might be encountered within the COL FOR . . . NEXT loop (though it shouldn't be in a properly written-out file), it is a good idea to include an IF statement telling the program to EXIT the DO WHILE operation. The EXIT command tells the computer to go directly to the next statement following the loop (in this case, a RETURN from the subroutine).

Either method works perfectly well. The first (printing the number of records in the file) lets your program know instantly how many sets of items there are. The second (use of an end-of-file marker) lets you create sequential files that don't have any extraneous characters until the end.

ADDING INFORMATION TO THE END OF FILES: APPEND

There are times when it's useful to tack additional items onto the end of an existing file. For example, a consultant who works by the hour might want to build a file that documented the length and content of business-related phone conversations. Data about each new conversation could be *appended* to the end of the file. Then, at the end of the month, the consultant could load the file into a word processor and proceed to type comments on each item.

Naturally, you could read the entire file into an array, add one more item, and then rewrite the entire file. But there's an easier way: the APPEND command. Figure 5-9 shows how APPEND works.

What if you could glue information to the end of the file? That's what APPEND is all about. The

1. Open the file.
2. Read in the information using INPUT or other commands:
   - Data must be read in exactly the same order it was written
   - Input the number of items first, if it was so written
3. DCLOSE the file to "free up" the file number.

Fig. 5-8. Steps for reading a file.

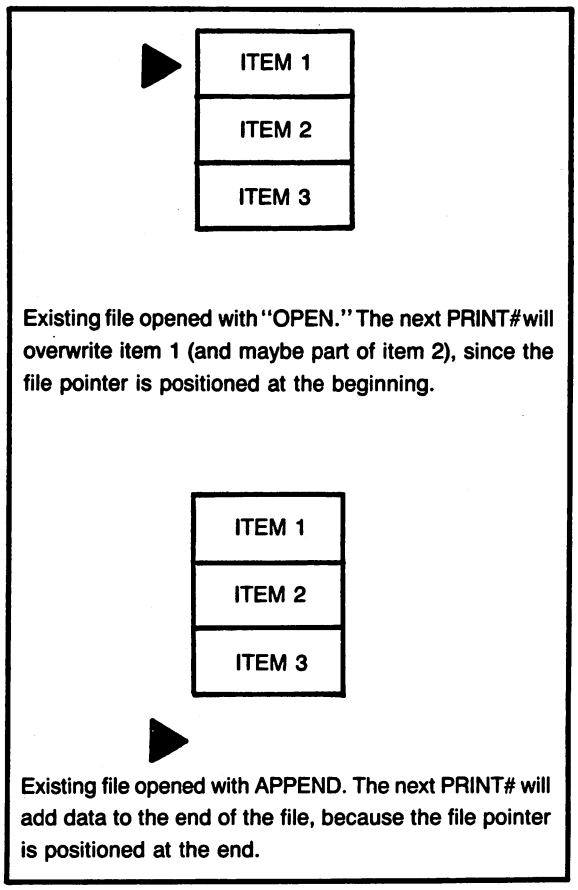


Fig. 5-9. How APPEND works.

APPEND command allows you to add data directly to the end of a sequential file using the APPEND command. APPEND follows the same format as the DOPEN command:

```
6005  APPEND#1,"MY FILE"
```

Because APPEND by definition always involves writing to a file, there's no need to include a ,W parameter. Since APPEND doesn't actually replace the file (it adds to it), it is not necessary to include the @ replace symbol before the filename. The commands to place information in the file are the same as before:

```
PRINT#1,"SPAGHETTI":DCLOSE
```

Just remember that when the file is opened with APPEND, all information is written to the end.

When Not to Use APPEND

It's a bad practice to use the APPEND command in programs that are used primarily storing and recalling records.

First, there's no way the program can tell how many records in the file once new records have been appended, since the number-of-records variable is at the beginning of the file, and this section is not affected by APPEND. The only way around this is to include a second file that always contains the number of records that exist in the first file—and that can get complicated.

Second, if your filing program changes any data in memory—either by modifying records or deleting them—these changes won't be included in the file unless the entire array is rewritten on the disk. It's therefore a good practice to rewrite the entire file each time data is saved, there always being the chance that something has been modified.

If you're designing a filing system, it's generally best to rewrite the entire file on the disk again, adjusting the number-of-records variable to include the new items.

COMBINING SEVERAL FILES: THE CONCAT COMMAND

There are times when it's useful to glue two, three, or more files together, like different cars on a freight train. The CONCAT command glues together (*concatenates*) one file onto the end of another one. You can use several CONCAT commands in a row to combine several small files into one large file. This is especially useful, say, when you want to combine separate letters or reports for transmission via modem: your word processor may not be able to read in several very large files, but CONCAT combines files effortlessly. This example:

```
CONCAT "INSULTS" TO "INJURIES"
```

would add the information in INSULTS to the end of the INJURIES file. The INSULTS file would re-

main on the disk untouched, but the new size of INJURIES would be equal to the combined size of the original INSULTS and INJURIES files.

CONCAT is one of those commands that immediately sends control of the computer back to you. While the disk is whirring along, you can perform other nondisk-related procedures such as listing your program or typing information.

SOME BAD NEWS ABOUT INPUT AND A SOLUTION

As we've seen them so far, read and write routines work pretty well. But we haven't added any of the daily complications you're sure to encounter in the real world.

Once you have a working write-to-file routine that is placing information into the file, and once your read-the-file routine is inputting that information, the most common problem you'll encounter is one of missing information:

```
MISSION: IMPOSSIBLE
Friends, Romans, countrymen
```

will become:

```
MISSION
Friends
```

The reason for this is that both the INPUT and INPUT# commands typically ignore any information that follows commas or colons on a line. The more sophisticated your data handling and storage requirements, the more likely you are to need these characters. Could you imagine a word processor that didn't allow commas or colons? What about an address list program that didn't allow you to place a comma between city and state?

Some computers include a LINE INPUT command in their BASICs. In these versions of BASIC, LINE INPUT allows entry with commas, colons, quotes, and other symbols that cause trouble using the regular INPUT command. The C-128 has nothing like it.

The good news is that there are at least two BASIC programming solutions to this dilemma. They all require a little extra work.

Inputting with Quotes

To understand the first way around our comma and colon problem, it helps to know a secret about the INPUT command: INPUT accepts an entire line of information (commas and colons included) if the line begins with a quotation mark. So while inputting

```
MISSION: IMPOSSIBLE
```

produces only MISSION, typing "MISSION: IMPOSSIBLE works just fine. A quotation mark in front of any line works as a *flag*, telling INPUT to accept the item literally, and to include the commas and quotation marks as part of the string.

You'll notice right away that this routine has a special requirement. All information in the file must be preceded by quotation marks.

But how do you set quotation marks in front of each file item? First, you and you alone control how the data is written into a file to begin with. So you can design the program to place a quotation mark in front of each line as that line is written into the file.

Along comes another hitch: you can't directly tell the computer to print a quotation mark, either onto the screen or into a file:

```
6030 PRINT#1,"A$ :REM THIS WON'T
                    WORK
```

The above line won't print a quotation mark, nor will it place the value of A$ in the file. Instead, the computer will take the line literally and print the letter A and a dollar sign (remember that anything following a quotation mark in BASIC is printed character for character).

Fortunately, there's another way to print quotation marks and other symbols: ASCII codes (see Fig. 5-10). Each character that can be displayed or typed in can be represented by special codes using the CHR$(x) function, where x is the ASCII number. Most computer users are aware that each keyboard character has a special computer code. But

32	SPACE	60	<	88	X
33	!	61	=	89	Y
34	"	62	>	90	Z
35	#	63	?	91	[
36	$	64	@	92	£
37	%	65	A	93	]
38	&	66	B		
39	'	67	C		
40	(	68	D		
41	)	69	E		
42	*	70	F		
43	+	71	G		
44	,	72	H		
45	-	73	I		
46	.	74	J		
47	/	75	K		
48	0	76	L		
49	1	77	M		
50	2	78	N		
51	3	79	O		
52	4	80	P		
53	5	81	Q		
54	6	82	R		
55	7	83	S		
56	8	84	T		
57	9	85	U		
58	:	86	V		
59	;	87	W		

Fig. 5-10. A partial listing of ASCII character codes.

many programmers shy away from employing these codes in their programs because the numbers are difficult to remember and program lines using them appear complicated. An ASCII code is just what the doctor ordered when you're trying to place a quotation mark (") at the beginning of a line. A quotation mark is ASCII code 34, and the character (CHR$(x)) function is used to print it on the screen or to a file:

```
10 PRINT CHR$(34)
```

It is also possible to define a string variable as a quote, and then to use this variable whenever a quote is required:

```
2 QUOTE$=CHR$"(34)   :REM ASSIGN QUOTE VALUE TO QUOTE
. . .

6030 PRINT#1,QUOTE$+A$  :REM THIS WORKS BEAUTIFULLY
```

Essentially, line 6030 adds a quote to the left of the string. BASIC prints both together in the file. A file written by this routine might look like this:

```
"Famous quotations:
"Friends, Romans, countrymen: lend me your ears
"I came, I saw, I conquered
```

Because of the leading quotation marks, BASIC's INPUT# function would reach each of these lines with commas and colons intact. The quotation marks are dropped automatically from the beginning. A major advantage is that the input file routine we covered earlier would remain completely unchanged.

Another Curve in the Road

The quotation mark approach works well, as long as you don't actually want to place quotation marks within a line. Except for the use of quotes at the beginning of a line, BASIC's INPUT# is not capable of reading quotation marks; in fact, INPUT# hiccups whenever it hits a quote in the body of the line. The result is either a "bad data in file" message, or a completely scrambled version of the line you're trying to read from disk.

As always, there's a fairly simple solution that can be called as a subroutine right before data is written to or read from your file. The trick in this case is to temporarily transform quotation marks into something else—say, a seldom-used graphic symbol. The data can be decoded as soon as it is read out of the file again. From a programming point of view, it's a little more complicated than some other solutions we'll talk about, but the encode/decode approach is very, very fast. The routine in Fig. 5-11 uses BASIC's lightening-quick INSTR function to scan the line for quotes. When

```
0 :
2 : rem  scan for quotes/convert
3 :
6 qt$=chr$(34)  :rem define quote
7 qr$=chr$(255) :rem use pi as quote replace
9 do
10 input a$
20 gosub 6200
30 printa$
40 loop
100 :
200 :
6200 : rem  code quotes
6205 : dun=0:place=1
6207 :
6210 : do until dun
6215 :    place=instr(a$,qt$,place+1)
6220 :    if place then gosub 6300: else dun=1
6230 :    loop
6235 :
6240 : return
6250 :
6300 : rem switch to graphic chartr
6305 :
6310 : a$=mid$(a$,1,place-1)+qr$+mid$(a$,place+1)
6320 :
6330 return
```

Fig. 5-11. Scanning lines for quotation marks.

a quote is found, a subroutine within this routine converts it to a graphic symbol, so that:

The doctor said "hello", and everybody waved.

becomes:

"The doctor said πhelloπ and everybody waved.

The only change in the write-to-file routine would be a call to the scan-for-quotes subroutine:

```
6010  FOR ROW=1 TO LAST
6020  :  FOR COL=1 to 3
6025  :  GOSUB 6200  :REM ENCODE FOR
         QUOTES
6030  :  PRINT#1,QUOTE$+
         FOOD$(ROW,COL)
          . . .
```

Note that while we're taking out any quotes that may be in the body of a line, we are placing a quotation mark at the beginning of the line as it is input (using QUOTE$). This ensures that commas and colons will continue to be accepted.

Figure 5-12 shows how this line would be reconverted (decoded) once it is read back in from the file. In reading the file, the decoding routine would naturally be called after the line is input from disk:

```
7010  FOR ROW=1 TO LAST
7020  :  FOR COL=1 TO 3
7030  :  INPUT#1,FOOD$(ROW,COL)
```

```
7035 : GOSUB 7200 :REM DE-
       CODE/PUT QUOTES BACK IN
  . . .
```

As long as you have control over the format of the files you'll be reading and writing, the procedure we've been talking about is fast, efficient and works very nicely.

Using the GET# Command

There are occasions, however, when you must maintain compatibility with files created by other programs. For example, your program might need to read files created by a word processor. It would be unrealistic to expect files from the outside word to start all lines with quotation marks; in fact, it's highly unlikely that they will.

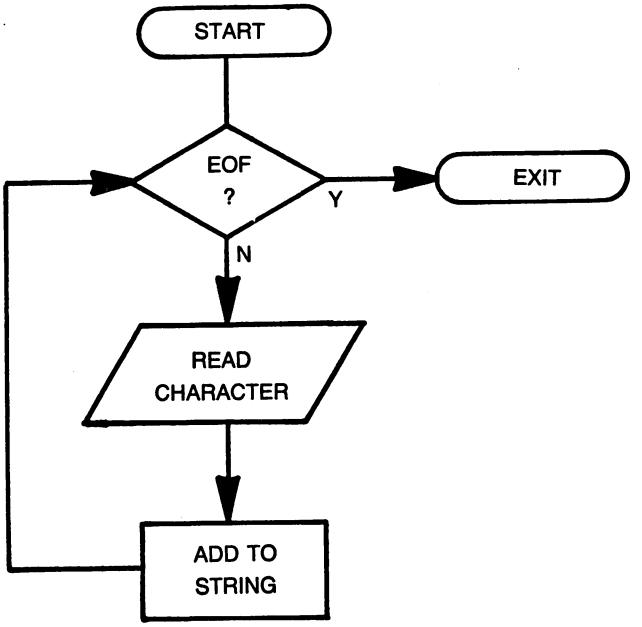
The best way to recall such files—and be assured of not missing any commas or colons—is to read each line one character at a time. This isn't as difficult as you might think, since the routine to read the line character by character can be neatly tucked away from your read-file loop.

The GET# command retrieves single characters from a file, and thus is ideal for this type of operation. The routine to read a line of data looks like this:

```
7300  IN$=" ":GT$=" "    :REM INITIALIZE
                          IN$
7305  DO UNTIL GT$ =      CHR$(13)
                          :REM CARRIAGE
                          RETURN
7308    IN$=IN$+GT$       :REM "BUILD"
                          IN$ WITH GT$
```

```
0 :
2 :
3 :
6 qt$=chr$(34)  :rem define quote
7 qr$=chr$(255) :rem use pi as quote replace
9 do
10 input a$
20 gosub 7200
30 printa$
40 loop
100 :
200 :
7200 : rem  decode/extract for quotes
7205 : dun=0:place=1
7207 :
7210 : do until dun
7215 :    place=instr(a$,qr$,place+1)
7220 :    if place then gosub 7300: else dun=1
7230 :    loop
7235 :
7240 : return
7250 :
7300 : rem switch back to quotes
7305 :
7310 : a$=mid$(a$,1,place-1)+qt$+mid$(a$,place+1)
7320 :
7330 return
```

Fig. 5-12. Reinserting quotes into a converted line.



Write each pass through the loop, an additional character is read and added to the string:

PASS #	STRING
1	T
2	TE
3	TES
4	TEST

Fig. 5-13. How a string is built one character at a time in a GETKEY loop.

```
7310  GET#1,GT$        :REM GET SIN-
                        GLE CHARACTER
7320  LOOP             :REM KEEP
                        GOING
7330  RETURN           :REM END OF
                        ROUTINE
```

The CHR$(13) in line 7305 is the ASCII symbol for a carriage return, which always signals the end of a line in sequential files. This is the only way to test for a carriage return. When using the INPUT or INPUT# commands in BASIC, you may have seen constructions used to test for carriage returns such as:

```
INPUT A$: IF A$=" " THEN PRINT "YOU
PRESSED RETURN ONLY"
```

With INPUT, an empty string (A$=" ") indicates a carriage return. This does not work, however, with GET#. Because GET# is pulling individual characters from the file, each GT$ in this example will contain a character code—even invisible characters such as carriage returns.

Figure 5-13 shows how a read character-by-character routine would work. The IN$ string will contain an entire line of text when the operation is finished. The content of GT$, a single character, is simple added to the right side of IN$ at each pass. Note that IN$ is built before the routine sets the next character, ensuring that a carriage return is never placed at the end of IN$ (because when a carriage return in encountered, the loop automatically ends). It's important that a carriage return not be placed at the end of a string, because this

will affect how it's displayed and stored.

The position of line 7308 in the loop means that the first time through, GT$ is added to IN$ before the first GT$ has actually been read from the file. This is okay, though, because on the first pass, both GT$ and IN$ are empty strings (equal to " ").

Using this approach, the write-to-file routine doesn't change from our first example. Because GET# accepts commas, colons, and all other characters automatically, there's no need to precede lines with quotation marks.

Chapter 6

Relative Files

If you've already overcome your fear of using sequential files, such as those discussed in previous chapters, then *relative*, or *random access*, files will be a cake walk—as long as you follow the rules.

But graduating to *relative*, or *random-access*, files is akin to transferring to a military academy; if you don't follow the rules from the beginning, you're bound to get into lots of trouble.

WHAT ARE RELATIVE FILES?

There's an old analogy that compares random-access files to phonograph records and sequential files to cassette tapes. Tapes are fine, the story goes, when you want to listen to an entire album. But tapes aren't so good for picking out an individual song—you might have to scan the entire cassette to find the tune you're looking for.

LPs, on the other hand, are perfect when you want to play a particular cut; you simply drop the needle (gently!) at the beginning of the band. When the song is over, you can start it again in a matter of seconds by picking up the tone arm and positioning it again at the beginning of the band.

Relative files allow you to do the same thing on a disk. They're particularly useful for address lists, inventory systems, and other applications where you need to quickly retrieve specific records of information.

But relative files have many other advantages. In addition to being faster than sequential files, they're also safer. Because each record can be written to the disk individually, programs using relative files generally save the record as soon as it is entered from the keyboard and verified by the typist. This means there's much less chance of loss due to power outages or accidental shut-off of the computer. With sequential files, these dangers always exist because a sequential file cannot be written back to the disk until all information has been updated or added in memory (unless you want to rewrite the sequential file each time a record is added or changed, which would be painfully slow).

Operational Differences

When you look at relative files for the first time, they'll no doubt seem constrictive.

First, because of the way the disk operating system organizes and accesses relative files, each record must be the same length. It's as though each song on our LP had to last for exactly the same number of minutes and seconds. This means you must know ahead of time exactly how much data you plan to cram into a record—and you can never exceed this number of bytes (characters).

You should also know the exact length for each field in a record, because the Commodore 128 handles relative files best when each item is assigned to its own predetermined field. Figure 6-1 shows how relative records are arranged.

On top of all of this, you must also know all of the operation information required for sequential files: how many records there are and how the items are arranged.

The section on input routines in the next chapter shows how you can prevent the entry of fields that are too long. This chapter also contains an input routine that does the same thing, though not as elegantly.

PLANNING RELATIVE FILE RECORDS

It's not enough to have a rough idea of what you want to do with a relative file. You should plan each file on paper before you begin. Here's what you'll need to know:

1. The number of items (fields) to be placed in a record.
2. The length of each of these items.
3. The approximate number of records you will eventually store in the file.

The first two items of information are necessary for proper planning of your records. The third will be used in setting up dummy records on your system. These blank dummy records aren't absolutely essential, but they do ensure that your file-handling program will work faster and more efficiently.

Planning Items in a Record

Before you give a speech, it's a good idea to know what you're going to say. Before you sit down at the keyboard to design a relative file system, it's essential that you know what information you wish to keep track of. The process of adding an extra item is quite tedious after a file has already been created.

Record Length	Records on a Disk
8	20,890
16	10,444
32	5,221
64	2,610
128	1,304
254	652

The number of records that will fit on a disk is the same for both single- and double-sided formats. Because relative files can occupy only one side of a disk, you can place programs and other files on a double-sided disk without compromising storage capacity.

Fig. 6-1. The number of records on a disk.

FIRST NAME	10
LAST NAME	20
HOME PHONE	13
OFFICE PHONE	13

Above is a typical file layout with field lengths for a telephone list. The above listing defines a small record. A larger set of data, such as are required by a full-fledged address program, would be laid out in much the same fashion.

Note that the length of a single field should not exceed 88 characters, because of the limitations of the INPUT# command with which you will be reading this information. (The only way around this ceiling is to use GET# to retrieve characters individually from a record. If you plan to design such as routine, be sure to allow time afterward for an hour of meditation or a good stiff bourbon.)

Record Size

Once you've determined the names and lengths of your fields, a record size can be easily determined. Simply add up the lengths of all the fields, and then add one byte for each item to allow for a *buffer separation character.*

In our previous example, a preliminary length would be determined by adding 10 + 20 + 13 + 13 to give a starting length of 56. We would then add 4 to this number (one byte for each of the four items in a record), to derive a total record length of 60.

The result of 60 means that when the record is opened for the first time using the DOPEN# command, the length specified must be at least 61 characters.

Some programmers like to add a few additional bytes to the record length in case they later want to add another field, but this is not necessary. There is another reason, however, that you might want to make a record slightly larger than is required. The reason is speed.

Making a Record Bigger for Speed. When the Commodore 128 is told to read records from a disk, or write records to it, the disk operating system places records in a holding pen known as the *disk buffer.* When dealing with records of 64 bytes, four records would fit in the buffer at once. Translation: The computer would read records four at a time.

Because all information is recalled and written in *blocks* of 256 bytes (characters), it's most efficient from a speed standpoint to read or write records whose lengths are evenly divisible into 256. If you use odd-length records, your programs will still work, but you won't be squeezing every ounce of stamina out of your disk drive.

Don't worry if you don't follow the reasons. The bottom line is that rounding the size of record upward can improve access speed. Here are the best record lengths to use:

```
  8 bytes per record....(32 records per block)
 16 bytes per record....(16 records per block)
 32 bytes per record....( 8 records per block)
 64 bytes per record....( 4 records per block)
128 bytes per record....( 2 records per block)
254 bytes per record....( 1 record  per block)
```

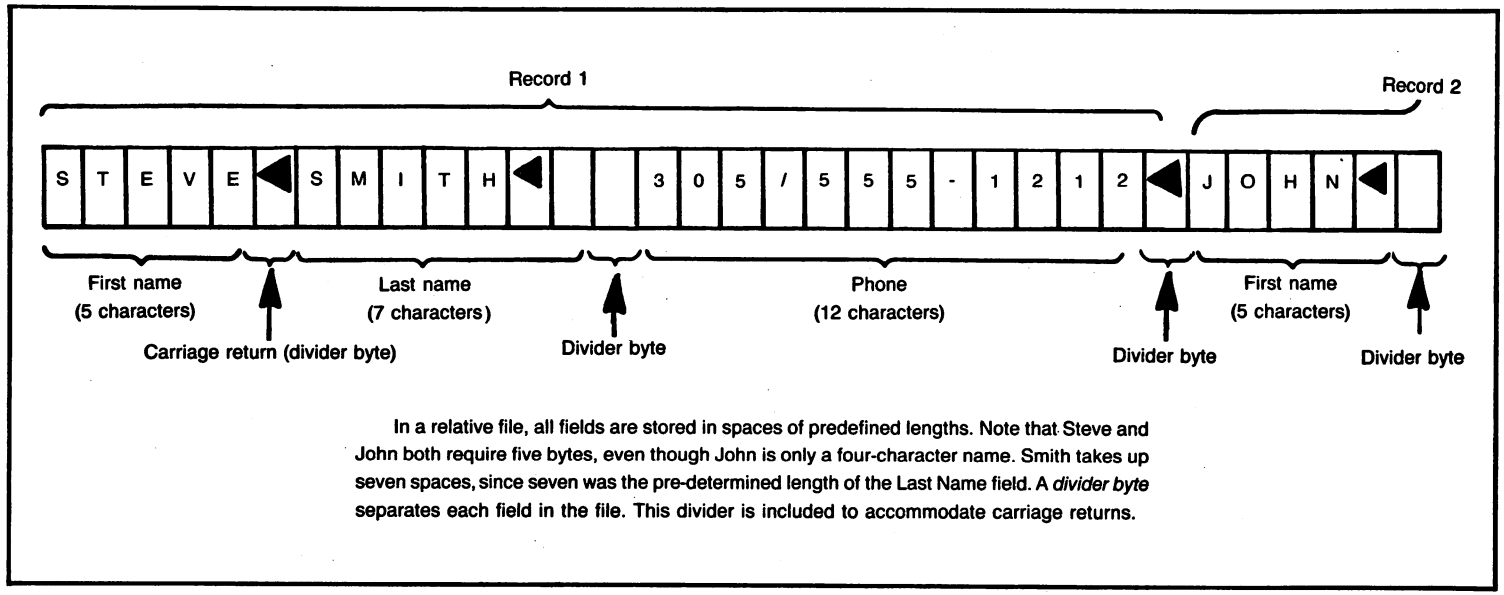
If you've scanned the section on relative files in your disk drive manual, you already know that a record cannot be greater than *254* bytes in length. *Two-hundred fifty-four* characters is a lot more than you think; you will probably never find this limitation a hindrance. If ever you do, you can redesign your subroutine to read and write sets of records at a time. The number of records that will fit on one disk depends on the number of bytes in each record. See Fig. 6-2 for further information.

The Number of Records

Another processing trick is to create a predetermined number of blank records. This isn't as difficult as it sounds, since a simple C-128 command does most of the work.

Why does this speed writing and recall of records? Simply because DOS usually stores records in any available nook or cranny of the disk. When individual records are stored in this manner, a file tends to become scattered all over the disk, and the drive has to work overtime retrieving them.

But when all records are created at the same time, especially on a recently formatted disk, all the



In a relative file, all fields are stored in spaces of predefined lengths. Note that Steve and John both require five bytes, even though John is only a four-character name. Smith takes up seven spaces, since seven was the pre-determined length of the Last Name field. A *divider byte* separates each field in the file. This divider is included to accommodate carriage returns.

Fig. 6-2. How relative records are arranged.

1. Defined file names and length.
2. Add up all lengths.
3. Round record size upward, if desired.
4. Determine the number of records per sector and add one (you can do this in the program).
5. DOPEN the file, and attempt a "read" of the record number determined in Step 4.
6. If the record does not exist, create the file by printing a null in the last record you anticipate using (record 100, 150, 200, etc.).
7. Update the *tracking record* (#1) with a zero, since no data currently exists in this new file.
8. DCLOSE the file to ensure that the 0 will be written to record 1.
9. Reopen the file,and return to normal program operation.

Fig. 6-3. Steps to create a relative file.

data is stored together on the disk. The drive head has to do much less moving around.

A PRACTICAL EXAMPLE

The easiest way to learn something is by doing it. In the next few pages, we'll construct a simple relative file program using the telephone list outlined previously. Figures 6-3, 6-4, and 6-5 outline the steps required to create, write to and read relative files on the Commodore 128. Through related program listings, you'll be able to see how each of these operations works.

Opening Assignments

Without a doubt the hardest part of using relative files is the setup process itself. The opening variable assignments that guide the creation and use of the file require some simple addition exercises and some forethought. If you've already planned out the file, though, writing this part of your program can take less than five minutes.

Figure 6-6 shows a typical subroutine used to assign the variables required to fuel a relative file. The first array of variables, designated by the name HEAD$, is intended to contain field headings for each of the items in a record. They don't have to be placed in this particular spot in the program, but situating them here, at the top of this routine, helps to clearly identify the purpose of each item in the record. These headings could be used as on-screen prompts during record entry and could also serve as heading labels on printouts.

Field Lengths and Item Sections. The next set of variable assignments handles the job of letting the system know about the length of each item and their beginning positions in a record.

The PLACE variables mark the exact starting

1. Open the file using DOPEN#, and specifying the length with the L parameter.
2. Read in the "Number of records" from record 1.
3. Position for file pointer at the proper record, using RECORD# command.
4. Issue the RECORD command again, positioning the file pointer at the specified record and item number.
5. Write the information for the item using PRINT#.
6. Continue this position-and-write procedure until all items in the record have been written.
7. Update the Number-of-Records record, if applicable.

Note: The first two steps need only be performed when the program is first run.

Fig. 6-4. Steps to write to a relative file.

1. Open the file using DOPEN# and specifying the length with the L parameter.
2. Read in the number of records form record 1.
3. Position the file pointer at the proper record, using the RECORD# command.
4. Issue the RECORD command again, positioning the file pointer at the specific record and item number.
5. Read the information for each item using INPUT# or a GET# loop.
6. Continue this position-and-read procedure until all items in the record have been read.

Note: The first two steps need only be performed when the program is first run. Only Step 5 differs from the procedure to read a record. All other steps are the same for both read operations and write operations.

Fig. 6-5. Steps to read a relative file.

points for each field. This PLACE array will be used as the *position parameter* in the RECORD# command when you are writing or reading records.

The LE array stores the actual length of each item as it should be entered from the keyboard. This number corresponds to the number in the original file layout discussed earlier, where the first name is ten characters, the last name is twenty, and the two telephone numbers are assigned thirteen characters each. The LE variable can be used not only in screening input, but in positioning items on printed lists and on the screen.

Notice how the PLACE variables are computed; they are the result of the last PLACE value,

```
60000 ::::::::::::::::::::::::::::::::
60010 rem           opening assignments
60020 head$(1)="first"
60030 head$(2)="last"
60040 head$(3)="home phone"
60050 head$(4)="office phone"
60060 :
60070 place(1)=1   : le(1)=10
60080 place(2)=12  : le(2)=20
60090 place(3)=33  : le(3)=13
60100 place(4)=47  : le(4)=13
60110 :
60120 lr=64        : rem length of rec
60125 fields=4
60140 file$="test"
60145 quote$=chr$(34)
60148 el$=chr$(27)+"q"
60150 :
60190 return
```

Fig. 6-6. Opening assignments for a relative filing program.

plus the length of the previous field, plus an additional character as a *dividing byte*. Thus PLACE(2) is figured this way:

```
12 = 1 + 10 + 1
```

And PLACE(3) is calculated with these numbers:

```
33 = 12 + 20 + 1
```

If you deal frequently with relative filing programs, you may want to design a short routine that will do these calculations for you automatically based on the length parameters, but the approach taken in our opening assignments example works fine and is easy to understand.

Record Length. The next variable assignment, LR, is used to set a record length. This variable name was chosen at whim; you could as easily use another one such as SIZE, or LNGTH. In fact, you can even use constant numbers (10, 35, 64) with the OPEN# command, dispensing with length entirely. The advantage of setting a variable is that you can set it in this subroutine, allowing your OPEN FILE subroutine to remain the same from program to program.

You'll note that a record length of 64 has been chosen—slightly larger than is really necessary for these records. As explained previously, rounding a record length upward to the next even divisor of 256 can often speed up the system's access to relative records. In this case, an original record length of 60 has been upped to 64.

Of course it doesn't always pay to expand the size of records in this manner. For example, if you'd had an initial record length of 35 bytes, you wouldn't normally round it upward to a length of 64; you'd be almost doubling the size of each record, and your file would take up almost twice as much disk space as necessary.

Other Setup Variables. The FIELDS variable at line 60140 determines the number of items in a record. It will be used in all program loops that read records from the disk.

The FILE$ variable defines the name of the file. QUOTE$ isn't used in our examples, but it could be; it's used to place quotation marks at the beginning of a line.

Finally, EL$ is an *escape* sequence used to clear a line from the current cursor position to the right. This variable can be printed or used as part of the CHAR command. We'll be using it to perform some tricks in an input routine.

With the variables in Fig. 6-5 properly set, you can use the remaining routines from this chapter on a "cookie cutter" basis, creating relative files for varying applications.

Opening the File

Any good open routine for relative files should also sense if the file does not exist. If it isn't on the disk, the program should create the relative file automatically.

The file open routine shown in Fig. 6-7 does all of these things with remarkable simplicity, thanks to Commodore's easy-to-use disk error codes.

If you could always be certain of the file's presence on a disk, and you always knew how many records there would be, the DOPEN# command in line 14030 would be sufficient to prepare the file for reading and writing. This is the same command used to open sequential files in some of our previous program examples. The difference is that here there's an extra part: the L (length) parameter.

The L parameter defines the length of this record, and by its very presence tells DOS that this is a relative file. Once the file has been opened for the first time using L, this parameter may be omitted. The Commodore will automatically know the length and file type on subsequent file opens.

Note that both the filename and record length are expressed as variables here. Because this open routine uses variables that were set in the opening-assignments section of the program, it can be used in any of your relative file programs without alteration. Just remember to set the FILE$ and LR variables.

As all variables used in the open command must be, the FILE$ and LR designators are enclosed in parentheses.

```
14000 ::::::::::::::::::::::::::::::::
14010 rem                      normal open
14020 :
14030 dopen#1,(file$),l(lr)
14032 :
14035 rcrd=256/lr+1:    rem next sector
14037 :
14040 record#1,rcrd:record #1,rcrd
14050 :
14060 if ds=50 then gosub 14100
14065 :
14070 record#1,1:record#1,1:rem # of recs
14075 :
14080 input#1,nr
14099 return
14100 ::::::::::::::::::::::::::::::::
```

Fig. 6-7. An open routine for a relative filing program. Normally, this routine is called only once. The call to 14100 occurs only if the file does not yet exist. (That routine creates the file.)

Creating a New File. Thanks to lines 14035, 14040, and 14060, the open routine appears more complex than it really is. Translated into English these lines would read:

14035. Figure out how many records fit in a sector and add one so we'll have the number of the first record in the second sector of the file. Store the number in a variable called RCRD.

14040. Position the file pointer for file number one (our currently open relative file) to the previously selected record number, which is stored in RCRD. Do it twice for good measure.

14060. If a disk error #50 occurs (meaning that the record doesn't exist yet), assume that this is a brand new file and call the routine for creating the necessary new records.

All of this has probably raised some questions. Why is it necessary to create new records? Why can't we just read the first record to see if there's anything there? Why does the program use the RECORD# command twice?

The answers all have to do with the peculiar way in which DOS operates.

We've already discussed to some extent the reasons for creating extra relative records: you're ensured that access will be faster and there's a guarantee that the program won't run out of room as it adds new information (the records are already there). The time to create these blank records is when the file has first been opened and no data has yet been stored, but how can you know if this is a new file or one that has been previously opened? The easiest way is by forcing a disk error by telling the computer to do something that is physically impossible—reading a record that does not yet exist.

Earlier it was mentioned that each block on a disk can hold 256 characters of information. The file buffer in your disk drive stores the same number. When a relative file is first created, DOS takes the liberty of making as many blank records as will fit in one sector, since at least one sector will be used for this file.

In short, reading the first few records of a file won't work—there won't be an error because these records have likely been created already when the record was opened. To force an error (if this is indeed a new file), we must position the file pointer at a currently virgin sector.

Line 14035 brings us to the question that you probably didn't ask out loud but were no doubt thinking. Why does the formula (RCRD = 256 / LR + 1) look so complicated?

Actually, it's quite simple. It divides the number of bytes in a sector by the number of bytes in

this file's records, giving the precise number of records that will fit in a sector. By adding one to this value, we have the number of the record that will start in the second sector of the file. If there is no record here, an error will occur and the program will know that it's time to setup a new file.

The error code that will occur is #50, "Record not present." This message won't actually appear on the screen, but the disk error variable DS will be set. After the error is encountered, processing will continue normally.

Line 14060 tests for the error condition and goes to a record-setup routine if the error is encountered. We'll explore this short routine in a few more paragraphs.

Tracking the Number of Records. The final few lines in this routine have to do with keeping track of the number of records in a file. As has been stated previously, the number of records should almost always be part of the file itself. This approach avoids all manner of confusion that might occur otherwise.

The best place to store this value is right at the head of the file. Many computer systems allow use of a record # 0, which is used to store the number of records. Alas, Commodore DOS has no such record, beginning its relative files with record number one. Even though this arrangement will cause us some brief programming headaches in some other routines, record number one is still the best place.

For technical reasons, it is recommended that all RECORD# commands be repeated when they will precede an INPUT# statement. If you don't follow this guideline, Commodore says, your data may become "corrupted". Once the relative file pointer is properly positioned at record one, the number of records can be input. If this file previously existed, the variable will now contain the number of records currently containing information.

Creating Blank Records Automatically

Figure 6-8 shows how a new file should be created. First, because the creation process takes a few seconds, a message should be displayed explaining to the user what the disk drive is busy doing. Line 14125 does this using the CHAR command.

Next, the record pointer should be positioned at the last record you anticipate using for this file. For small files, 100 is generally a good limit. Don't worry about underestimating; if you exceed 100 records, the additional information can be added to the existing file with no problems.

For the reasons outlined earlier, you should issue the record command twice.

Line 14140 prints a *null* character in record 100. Issuing the command in this manner forces the computer to create 100 blank records. There's no need for any loops or other code for this; DOS does it automatically.

```
14100 ::::::::::::::::::::::::::::::::::
14110 rem            create if not exist
14120 :
14125 scnclr:char ,1,12,"creating the "+file$+" file"
14130 record#1,100:record#1,100:    rem 100 records
14140 print#1,chr$(255): rem null record
14145 record#1,1:record#1,1:rem # of recs
14148 print#1,0:  rem no rcrds yet
14150 dclose#1
14160 dopen#1,(file$),l(lr)
14199 return
```

Fig. 6-8. A routine to create a new file.

The Number of Records

Since a good filing program always knows how many records there are, we must update this tracking record now that the file has been properly created. The current number of records is zero. This amount is written into the file with the following steps:

1. Position the record pointer at the first record, where we will store the current number of records. Do it twice for good measure. (line 14145)
2. Print a zero (0) in this record, so the program will know that it's starting out with no records. (line 14148)
3. Close the file using the DCLOSE# command. This is a precautionary measure ensuring that the zero will definitely take when it is read back in.

Finally, since this is just a subroutine within the main file-open routine, and since the main routine doesn't know that the file has been closed, line 14160 reopens the file using the same parameters issued earlier. The file is now ready for use.

Writing to the File

Now that our file has been properly set up, we're ready to get down to business.

Storing information in a relative file is in many ways similar to writing to the sequential files discussed earlier. The main differences are that with relative files the following rules must be followed:

1. Sets of information must be stored in records of preset lengths.
2. Within each record, items must be placed in the proper positions.
3. If this is a new record, the number of records value in record #1 must be updated.

You can quickly see that the preliminary steps in our program have taken care of most of these differences. The record length has been defined in the setup subroutine, and records of the proper lengths have been created in the open routine.

We've also taken care of the item position requirement in the setup routine, by creating a *table* of record positions using the PLACE variable.

The Write Routine

The write-to-file routine shown in Fig. 6-9 is a simple but effective means of writing information to records in relative files. This routine can be used in any program when you wish to store data in an array. The following variables are used going into and coming out of the routine:

REC—used in other parts of the program to refer to the files usable records (usable record #1 one is really the second record in the file, since the first record in the file is used to keep track of how many records total there are).
FIELDS—defined in the SETUP routine as the number of fields in a record.
PLACE(x)—also defined in setup, PLACE(x) determines the exact beginning point in a record for a particular item being read into the array.
A$(x)—stores the data to be written to the disk into an array.
ADDED—tells the routine whether this record is new (to be added), or one that already existed and is being updated. If ADDED is equal to −1, the record is new.

There are other variables that are internal to this routine and are reset each time the routine is called. The following variables are used internally by this routine:

RCRD—converts the REC variable into the actual disk record number, by adding one.
ITEM—keeps track of the current item being read from a record into the array.

The write-to-file routine begins by figuring out the number of the record to be written to. Why is this necessary? Because of where we're storing the number-of-records variable—namely in record number one. It means that each record of real data will be pushed up one notch. Thus, phone record one

```
12000 ::::::::::::::::::::::::::::::::
12010 rem                  write to file
12015 rcrd=rec+1
12020 :
12030 record#1,rcrd
12040 for item=1 to fields
12045 :   record#1,rcrd,place(item)
12048 :   print#1,a$(item)
12050 :
12060 :   char ,1,10: rem posit cursor
12070 :   print "writing item# "item" for record# "rcrd
12075 :   print a$(item)"       "
12080 next
12090 if added then gosub 12300
12299 return
12300 ::::::::::::::::::::::::::::::::
12305 :rem          write # of records
12310 :
12320 record#1,1:record#1,1
12330 :
12340 print#1,nr
12350 :
12360 return
```

Fig. 6-9. A routine to write a record to a relative file.

is really disk file record two; phone record two is really disk file record three; and so on. Requiring other parts of the program to keep track of this could become confusing indeed, so we solve the problem by performing a simple conversion (stepping up the value by one) in the read and write routines.

REC is the *virtual* record number used by the program to track our phone list. It could be used in entry routines, search routines, sort routines, and any other applications in which the record number must be specified. RCRD, on the other hand, is an internal variable that will only apply to the read and write subroutines. It won't be used anywhere else, and it will be reset to a value of one greater than REC whenever the program goes to one of these routines.

Writing the Data. Placing the data in a relative record is now simply a matter of correctly positioning the record pointer at the record specified, starting the loop, and printing each item into the record at its proper position.

Line 12030 sets up DOS to read the record value stored in RCRD. The #1 in the command specifies that this is the first file, and has no relation to the record number (keep this in mind as you're working with relative files—two separate record numbers in the RECORD# command can become confusing even for veteran programmers).

Actually it's not absolutely necessary to position the record pointer here, since the RECORD# command is also used with the write-items-to-record loop that is to follow. But it's good to get into the practice of issuing RECORD# as a *set up*

command. The main reason is that in reading data from a disk, RECORD# sometimes fails to work properly unless issued twice. It's easier to write all your routines using double commands, than to try to remember when RECORD# must be issued twice, and when it need not be.

For the FOR . . . NEXT loop, we'll use the same counter variable we've called upon previously: ITEM. Because of the way it is used, this variable will always contain the number of the item currently being read from the file. It will serve as a *marker* for the current place in the A$(x) array.

Line 12045 positions the file pointer to the specific place in the record where it should be, and line 12048 prints the data into this slot in the record.

Lines 12060-12075 print an updated message on the screen as each item is written to the file. Line 12080 finishes the loop.

New Record. Naturally, there has to be some way to determine when a record is new, so that the number-of-records record can be updated appropriately. If an existing record has been rewritten with new information, we don't want to write the number of records to record #1, because it's redundant; the program should already have the current number of records on file.

On the other hand, if the record is a new one, you most definitely should update the number-of-records record.

The ADDED variable serves this purpose well. In every filing program, there should be an add-a-record routine that does the following:

1. Accepts entry of information from the keyboard.
2. Sets the ADDED flag to −1, indicating that a record is being added.
3. Bumps up the number of records variable (NR) by one.

Our write routine assumes all of these things have already been done, if appropriate. The only thing we'll check for is whether the ADDED flag is set. If it is, the program will visit the subroutine that updates the number of records (starting at line 12300). If the ADDED flag is set to zero, the program will skip the update routine.

Using Quotes and Other Tricks. Previous chapters have discussed the use of quotation marks for enclosing data that will contain commas or colons. If you plan to lead each string with a quotation mark, simply define QUOTE$ = CHR$(34) at the setup portion of your program. Then, add this quotation mark variable to the line that writes the data:

```
12048 : PRINT #1,QUOTE$ + A$(ITEM)
```

If you anticipate using quotations marks in your files, you'll probably want to use the conversion routine discussed in Chapter 5. If you use quotation marks, be sure to add one more byte per item when computing record length.

READING FROM THE FILE

As you can see from a quick glance at Fig. 6-10, the routine to read information from a record follows the same general procedures as our write routine. The two differences are that data is being read using INPUT# (instead of being written using PRINT#), and that there is no update to the number of records, since nothing is being added.

A Quick and Dirty Input Routine

A filing program would be nothing without a means of entering information from the keyboard. The input routine pictured in Fig. 6-11 is nothing fancy, but it will suffice nicely.

In addition to allowing input of information, this routine checks for over-length variables (those that would not fit into the record's item boundaries), and rather elegantly handles their re-entry.

This routine also takes advantage of some of BASIC 7.0's high-octane commands, such as DO/UNTIL/LOOP and CHAR. It also uses some other tricks, such as printing keyboard *escape codes* to clear a line on the screen.

While this routine uses a lot of flags and other variables, its structure makes it pretty easy to decipher. The following variables come from other places in the program or are used by other parts of the program after this routine finishes with them.

```
13000 ::::::::::::::::::::::::::::::::
13010 rem                  read from file
13015 rcrd=rec+1
13020 :
13030 record#1,rcrd
13040 for item=1 to fields
13045 :   record#1,rcrd,place(item)
13048 :   input#1,a$(item)
13050 :
13060 :   char ,1,10: rem posit cursor
13070 :   print "reading item# "item" from record# "rcrd
13075 :   print a$(item)"       "
13080 next
13299 return
```

Fig. 6-10. A routine to read a record from a relative file.

As in the past, we'll refer to these as *external* variables:

ADD—determines whether or not the screen should clear (is this an added item with no existing data?).

FIELDS—you've certainly seen this one before; it's the number of fields in the array.

HEAD$(x)—the heading (or title) for each item in the array. These item titles were defined in the original setup routine for variables.

A$(x)—the variable which stores array elements as they are entered. This is also the variable used to write and read data to and from the file.

Much more numerous are the routines internal variables, which serve as flags, or switches, to monitor the progress of different events. There are flags to tell the computer if the information entered is correct. There are flags to tell the computer whether the last item entered was too long. There's even a flag that lets the machine know if this is the first time the item is being entered, or if it's a retry after a too-long entry.

All these flags serve a purpose: they allow the routine to be driven with DO/UNTIL/LOOP structures instead of more cumbersome GOTO branches.

The internal variables for this routine include the following:

OK—this flag indicates that the answer to the "correct?" prompt was "y" or "yes." It drives the DO/UNTIL loop that keeps the program asking "is this correct?" until the user answers "Y."

ITEM—the same variable we've used before to keep track of current item in an array.

FRST—indicates whether this is the first time this item has been entered (on this pass). This variable makes sure that the program doesn't skip the DO/WHILE loop for verifying variable length.

BIG—was the last item typed too BIG? If it was too long, the program will continue in the DO/WHILE loop for length, until the length decreases.

Y$—the input variable that stores the user's response to the "is this correct?" prompt.

These variables should give you some insight into the workings of the input routine. Of course, some of the tricks employed here bear closer scrutiny.

How the Input Routine Works. The input routine in Fig. 6-11 is basically three big loops with some fancy stuffings inside.

The first is the DO UNTIL OK loop, which keeps accepting entry of names and phone numbers until the OK variable is equal to −1 (which hap-

pens to be when the user answers "Y" to the "is this correct?" prompt a few lines down in the program).

Inside this first loop is the FOR . . . NEXT loop, which begins at line 4060. It's pretty standard. The ITEM variable starts out at one and continues until reaching the value of FIELDS, which represents the number of fields in a record. Of course, in between the FOR and the NEXT are a group of commands designed to accept and verify keyboard entry.

Commands Inside the FOR . . . NEXT Loop. The set of commands nestled inside the FOR and NEXT statements is performed with each pass through the loop.

The first command inside the loop is:

```
CHAR ,0,23,EL$
```

```
4000 ::::::::::::::::::::::::::::::::::
4010 rem                        input items
4020 :
4030 if add then scnclr
4040 ok=0: do until ok
4050 char ,0,1
4060 for item=1 to fields
4070 :  char ,0,23,el$
4080 :  frst=-1
4090 :  do while big or frst
4100 :    char ,0,item : rem pos item
4110 :    print head$(item);
4120 :    input a$(item)
4130 :    gosub 4300:  rem check length
4140 :    frst=0: rem no longer 1st
4150 :  loop
4160 next
4165 :
4170 char ,0,23,el$+"is this correct"
4180 input y$
4185 if instr("Yy",left$(y$,1)) >0 then ok=-1:else ok=0
4188 :
4190 loop
4200 return
4300 ::::::::::::::::::::::::::::::::::
4310 rem                     check length
4320 :
4330 if len(a$(item)) > le(item) then begin
4340 :      play "sceg"
4350 :      char ,0,23,"item too long -- please re-enter"
4360 :      char,0,item,el$
4370 :      big=-1      :rem         redo
4380 bend:else big=0
4390 return
4400 :
```

Fig. 6-11. A simple routine for the entry of records.

This command positions the cursor at column zero, line 23, and clears this line with an ESC-Q. The clear-to-end-of-line is necessary because some prompts and messages will be placed on this line during operation. It's important to clear these messages as soon as they're obsolete, so as not to confuse the program operator. For example, even if the last item was too long, you wouldn't want the same "item too long" message to continue to appear for entry of the next item; it should be cleared. So the CHAR command at line 4070 clears this line for good measure—even if there wasn't anything there.

Next, the FRST variable is set to −1, meaning that this first-time-through flag is on. Once the item has been entered from the keyboard once, the FRST flag will be set to off (0). Setting the FRST variable in this way makes sure that the DO/WHILE loop in the next line won't jump to the end of entry prematurely.

The DO/WHILE loop in line 4090 tells the system to keep asking for this item while it is too BIG, or when it is the FRST time through. The BIG variable will be set on or off in a subroutine that is called from within this loop.

The CHAR command in line 4100 positions the cursor at column zero on the line specified by ITEM. This means that the item one will be input on line one, item two will be input on line two, and so on. It's a poor man's answer to screen formatting—one we dispense with in the chapter on professional program appearance. For now, it'll be fine.

The PRINT statement in 4110 prints the heading at the current cursor position, which was specified by the CHAR command at line 4100. If you're observant, you'll notice that the program could have combined lines 4100 and 4110 into a command that looked like this:

```
CHAR ,0,ITEM,HEAD$(ITEM)
```

The two-line approach in our program routine is simply another illustration that the same effect can be obtained in many different ways.

Line 4120 asks for the item. Because there's a semicolon at the end of the last PRINT statement, the cursor is still next to this title; the information typed on the keyboard will appear directly to the right of the item title.

Finishing up the FOR . . . NEXT loop, we have a call to the subroutine at line 4300, which determines if the input string is too long, and sets the BIG variable accordingly. FRST is reset to zero, since at this point in the program we can guarantee that this string has been entered at least once.

Confirmation. At the tail end of the input routine is a prompt that asks the question "is this correct" (because the INPUT statement provides its own question mark, there's no need for one in the CHAR operation at line 4170).

Finally, line 4185 strips down the answer to the leftmost character entered and looks for a "Yy" matchup. If a match is found, OK is turned on (−1), and the condition for the beginning loop (DO UNTIL OK) is satisfied. If things are not OK, the routine goes back to the top for re-entry of the name and telephone numbers.

Other Routines Related to Input

The routine in Fig. 6-11 does a lot in a short space, but there's usually more required for file routines. Usually, separate add-a-record and change-a-record routines will call this main entry routine. This allows the program to display data where necessary and to set different flags depending on whether the record was added or is merely being changed. Figures 6-12 and 6-13 show typical routines. These routines use many of the same procedures called upon already in our input routine.

Adding a Record. The add-a-record routine in Fig. 6-12 performs the following functions:

1. It initiates a loop to continue adding records until it is no longer necessary.
2. It sets the ADD flag so the screen will be cleared before input.
3. It calls the input routine and accepts entry.
4. It bumps up the record counter by one, and assigns this new (highest) record number as the current record.
5. It calls the routine that writes to the records, and continues with the DO/UNTIL loop.

This process continues until the user answers "N" or "NO" to the "add another?" prompt.

Changing a Record. As shown in Fig. 6-13, changing a record is a bit more complicated, because information has to be retrieved from the disk and then displayed for review.

We start at line 3400, where a routine accepts a record number from the keyboard, and then calls the true edit routine, which does most of the work. This process continues until the user indicates that he doesn't wish to change another record.

The edit routine starting at line 3400 performs

```
3000 ::::::::::::::::::::::::::::::::::
3010 rem                    add a record
3012 again=-1
3015 :
3020 do until not again
3030 :  add=-1      : rem       add flag
3040 :
3050 :  gosub 4000  : rem      input rtn
3060 :
3070 :  nr=nr+1     : rem bump# of recs
3080 :
3090 :  rec=nr
3100 :
3110 :  gosub 12000: rem   write record
3115 :
3120 :  char ,0,23,el$+"add another"
3130 :  input y$
3135 :
3140 :  if instr("Yy",left$(y$,1)) >0 then again=-1:else again=0
3150 loop
3160 :
3165 added=0
3170 return
3400 ::::::::::::::::::::::::::::::::::
3410 rem                  change a record
3420 again=-1
3430 :
3440 do until not again
3450 rec=0
3460 :  do until rec >0 and rec<=nr
3470 :    scnclr:input"record (#)";rec
3480 :  loop
3485 :  gosub 3500       : rem  edit rec
3490 :  char ,0,23,el$+"change another"
3491 :  input y$
3492 :  if instr("Yy",left$(y$,1)) >0 then again=-1:else again=0
3494 loop
3496 :
3498 return
```

Fig. 6-12. Routines to accept an entry and add or change a record.

```
3500 ::::::::::::::::::::::::::::::::::
3510 rem                    edit a record
3520 :
3530 :
3540 gosub 13000: rem read rec
3550 :
3560 gosub 3800 : rem disply data
3570 :
3580 gosub 4000 : rem input rtn
3590 :
3600 gosub 12000: rem   write record
3610 :
3620 :
3630 :
3640 return
```

Fig. 6-13. A routine to edit a record.

the following simple steps:

1. Reads the specified record by calling the read routine.
2. Displays the data read from the file, by calling a display routine.
3. Inputs changes, by calling the input routine.

This edit routine could be called by a number of program options. For example, if you included a routine to search for records, you could call this routine when the records were located.

GOING FURTHER

The complete program in Appendix B pulls some different ideas together into a more sophisticated relative-files package. It employs some of the concepts discussed in other chapters, such as the use of more aesthetic screen format routines and menus.

Chapter 7

Professional Input Routines

Up to this point, you've probably gotten along very well with BASIC 7.0's built-in input routine. It's easy to use, but INPUT falls flat on its face when it comes to performing certain common entry operations—like accepting commas and colons (:) from the keyboard, responding to special single-character commands, or accepting lines that are longer than 88 characters.

OTHER WAYS TO INPUT DATA

You know by now, however, that with a little work, any obstacle can be overcome on the Commodore 128. This time Commodore has included two commands to help us out. They both accept characters from the keyboard one character at a time.

The two commands are:

GETKEY and GET

Both commands store a single keystroke in a specified variable. The differences between the commands are related to how they react when the key is pressed. GETKEY is the more patient command of the two. It will wait indefinitely for a key to be pressed. GETKEY is the command we'll use when first experimenting with keystroke input.

GET, on the other hand, simply checks once to see if a key has been pressed. If there's no character in the computer's *keyboard buffer*, GET proceeds on with the next operation. In order to use GET in a routine that waits for keyboard input, you must construct a loop that looks something like this:

```
CH$=" ":DO WHILE CH$=" ": GET CH$
: LOOP
```

The above approach is more complicated, but it has advantages too. The main use of GET is in routines where you'd like the program to be doing something else while it's waiting for a keystroke. An example later in this chapter will show how such a routine can be used to display the current time while the user is typing information.

The Limitations of the INPUT Command

In addition to its previously mentioned shortcomings, the INPUT command cannot be customized; you can use its built-in functions to your heart's content, but you can't use INPUT to set flags, screen keystrokes, or test other entry. If you want to assign a special function in your programs to, say, the ESC or TAB key, there is no way to do it with INPUT—nor is there any way to screen out letters or numbers in special fields; nor can field length be tested during entry.

INPUT does many things well, but these features just aren't part of the command.

The Silver Lining

For all its faults, input is a truly amazing command. It knows an awful lot about what's happening on the screen and what's being typed:

- It knows the current position of the cursor and where to place the cursor with each new keystroke.
- It knows how to insert or delete characters.
- It places everything you type into the proper variable.
- It knows when you have finished typing (when RETURN is pressed).

INPUT also performs some other tricks, such as scanning the currently displayed line for characters to be included in the entry.

A CUSTOMIZED INPUT ROUTINE

The input routines discussed in the rest of this chapter won't do every one of the screening and testing functions mentioned above, but they will provide a serviceable means of entering information. They'll also give you a lot of features not available through normal input.

Setting Up Variables

An input routine needs more variables than you might at first imagine; you've got to know where the entry starts, the entry length, whether you want to show characters in upper- or lowercase, and what kind of cursor you'll be using. The lines in Fig. 7-1 come from two sections of our input routine, and do all these tasks admirably.

The first variable on the list is NMBR, which determines whether this is a numbers only entry, and which is temporarily *remmed off* (turned into a REMark). If NMBR were on, the entry routine to be covered shortly would accept only numbers and a few other symbols.

Lines ten and twenty show the variables H and V, which set the horizontal and vertical (column and line) starting position of entry.

The variable LE sets the length for entry. If you were entering an array, this variable would be reset for each new item in the array, as would H and V.

While the variables at the top few lines will always be changing, the variables in lines 60000 - 60099 won't be. They are true setup variables that should not be reset at any time.

The lines here should be included as part of your general setup procedure for any program using the entry routine. Setup procedures for programs in this book generally begin at line 60000, but you can, of course, relocate the lines as required.

First, there is a musical variable, BLEEP$, used to give the computer's warning cry a little more pizazz. This variable will be used simply by issuing the command:

```
PLAY BLEEP$
```

Of course, for it to sound like a bleep—and not a melody—the notes will have to be played very fast, so TEMPO will be set to 255 at the beginning of the input routine, just in case a good strong BLEEP$ is needed. The play command is covered in more detail in another chapter.

Next in the variable list there is SPACE$, which is used in this routine to clear extraneous information from a line once the RETURN key has been pressed. Actually, you may find this a useful variable to include in all your routines, since it can be used to clear away undesired junk from the screen quickly.

The first variable is CSR$, which defines what the cursor will look like. In this case, we've chosen a thick underline, which is produced by pressing the Commodore (C=) and P keys. Because this graphic symbol is produced using the Commodore key, it is immune to the uppercase/graphics shift that occurs when you switch from the uppercase-and-graphics mode to lowercase-and-uppercase mode.

The second variable here, CASE$, determines how typed characters will appear on the screen. If CASE$ is equal to ASCII character 142, all typed characters will appear in the uppercase-and-graphics mode. If CASE$ is ASCII character 14 instead, typed characters will appear in the lowercase-and-uppercase mode. An example in the body of the entry routine will show how upper- or lowercase characters can be screened or converted.

Finally, there's RESET$, which is used to ensure that the screen doesn't get stuck in the infamous *quote mode*, where some keyboard characters will appear in inverse on the screen instead of being properly interpreted. If you've ever tried editing a program line after you've pressed INSERT or typed a quote, you've experienced the quote mode. It's no fun.

How the Entry Routine Will Work

The theory of accepting information from the keyboard is simple:

1. Accept characters one at a time until RETURN is pressed.
2. Build a string based on the characters typed.
3. Allow for backspacing.
4. Screen any undesired characters and block the entry of lines that are too long.

The execution is a little more difficult, because there are lots of tests to be performed along the way. To make it easier to read, most of our routine is *driven* by flags, which indicate to the computer exactly what type of data has been typed and tell the program how to react.

```
5 rem nmbr=-1
10 h=0
20 v=10
25 le=15
30 scnclr
35 char,h,v,"xxxxxxxxxxxxxxxxxx"
40 gosub 60000
50 gosub 5000
60 rem printin$
65 rem if esc then print "escape was pressed"
70 end
          . . . . . .
60000 :
60094 bleep$="s v1o5 g v2o6 g v304 g"
60095 space$="
"
60096 space$=space$+space$+space$
60097 csr$="_"
60098 case$=chr$(142)
60099 reset$=chr$(27)+"c"
61000 return
```

Fig. 7-1. Initial settings required for the input routine.

Variables Used Inside the Input Routine. There are few external variables used in this routine, other than the setup variables already discussed. The variable IN$ returns whatever has been entered from the keyboard. V, H, and LE have been previously discussed.

There are a lot of internal variables, because there's a lot to keep track of. In main processing routine (Fig. 7-2), the following variables are used:

BKS This flag is set by a subroutine any time a backspace character is pressed.

DUN Indicates that you are done with entry of this line. This flag is set in this routine if you press RETURN or ESCape.

CH$ Holds the last character typed. This character is then analyzed and placed in the string IN$, if appropriate.

CH This variable contains the ASCII value of the typed character. The reason we convert the string CH$ into a numeric variable is that operations involving numeric variables are much faster; it speeds up the entry routine.

IN A flag used to indicate whether the character typed should be included in the entered string (IN$). If RETURN, ESCape, or a backspace character was typed, IN will be zero.

LN This variable is set in a subroutine and contains the current length of IN$. It is used to test to make sure that the entry has exceeded the allowable length (stored in LE).

These variables are the core of the entry routine. As you can see from Fig. 7-3, almost every line of the routine involves at least one of them.

Let's take a minute to review the two main components of this subprogram. One is a main processing routine, which contains the loop and builds the IN$ entry variable as characters are typed. The other major routine screens entry and sets flags based on the character typed.

The Main Processing Routine. The main

```
5000 ::::::::::::::::::::::::::::::::::
5020 rem                    input routine
5040 :
5050 tempo 255
5060 gosub 5900          : rem clr buffer
5080 in$="":ch$="":in=0:dun=0
5085 :
5090 if le>240 then le=240
5100 :
5120 do until dun
5130 :  char ,h,v,case$+in$+csr$+reset$
5140 :  getkey ch$
5160 :  ch=asc(ch$)
5180 :  gosub 5400       : rem   test ch
5190 :  if dun then exit
5200 :
5210 :  if bks then bks=0:gosub 5950
5300 :  if in then in$=in$+ch$
5390 loop
5392 :  char ,h,v,case$+in$+" "+reset$
5395 return
```

Fig. 7-2. The main entry routine.

```
5400 :
5410 dun=0:in=-1:bks=0:esc=0
5415 :
5420 rem                    test character
5430 ln=len(in$): rem cur len
5440 if ch=13 then dun=-1:in=0: gosub 5800: rem cr
5460 if ch=27 then dun=-1:in=0:esc=-1:rem esc
5480 if (ch>144 and ch<149) or ch=157 or ch=95 or(ch<32)thenbks=-1:in=
     0:goto 5530
5500 if ch>96 then ch=ch-128:ch$=chr$(ch): goto 5520
5505 if nmbr and (ch>57 or ch<45) then play bleep$ :in=0:goto 5530
5520 if ln=>le then play bleep$:in=0
5530 return
5800 ::::::::::::::::::::::::::::::::::
5805 rem          clear to end of field
5810 :  char ,h,v,case$+in$+left$(space$,le-ln)+reset$
5820 :
5830 return
5900 :
5905 rem                      clear buffer
5910 forzz=1 to 8:get ch$:next
5920 return
5950 :
5960 :rem                  backspace rtn
5970 if ln=0 then tempo 255: play"s v1o5 c v2o6 c v3o4c":goto 5995
5980 in$=left$(in$,ln-1)
5995 return
```

Fig. 7-3. Routines to test characters and clear fields and buffers.

processing routine, shown in Fig. 7-3, starts by calling a short loop that clears the keyboard buffer—the repository for keystrokes that have not yet been accepted into a program.

The buffer should be cleared if you want to make sure users don't type ahead of the prompts on the screen. If you don't mind their typing ahead, this line can be eliminated, or *remmed off*. You can even include a flag here that would allow you to specify (in the routine that calls this one) whether the type-ahead buffer should be cleared.

Next we have line 5080, which clears out (*initializes*) some of the beginning variables. Of special note are IN$, which will contain the entire typed line, and CH$, which will store the value of each character typed.

Line 5090 is included as a safety feature. It limits the entry length to 240 characters, so you won't exceed BASIC's string size limit.

Now we move into the loop that really drives the routine. There's a lot of action packed into line 5130, which gives the user the illusion that a cursor is waiting for entry, and that each character typed magically appears on the keyboard. In fact, the innards of the routine are a little more complicated.

Basically, this line prints the current value of IN$, followed by the cursor character. During the first pass through, before any characters have been typed, IN$ will be blank, and all that will appear is the cursor symbol.

The expression on line 5130 is couched within

two other strings: CASE$ and RESET$. CASE$ was defined in the setup section to shift displayed characters into an uppercase mode. RESET$ is used to make certain the entry routine doesn't get stuck in the 128's dreaded quote mode.

The H and V variables in line 5130 position the string at the original starting position. So the entire variable IN$—plus a cursor—is printed each time a character is typed or backspace is pressed. This approach sacrifices a few nanoseconds of speed on very long strings, but it's the easiest and neatest way to display the typed characters.

Once the current value of IN$ has been displayed, the program moves on to lines 5140 and 5160, which accept a single character from the keyboard (CH$) and then derive the ASCII value of this character (CH). This may seem a roundabout approach to things, since the computer is perfectly capable of performing tests on string values, like this:

```
IF CH$ = CHR$(13) THEN DUN = -1 :REM RE-
    TURN WAS PRESSED
```

There's only one problem. When you're dealing with entry routines, speed is everything, and string comparisons are a lot slower than numeric ones. Rewritten into a numeric comparison, the above statement would look about the same:

```
IF CH = 13 THEN DUN = -1: REM RETURN
    WAS PRESSED
```

It would, however, execute many times faster. And when you plan to perform a number of different tests on a keystroke, the little delays add up.

The Rest of the Loop. The routine which actually tests keystrokes is tucked away from the main loop. This is primarily done for readability, since these test-character routines can take on the appearance of a bowl of spaghetti if they're written for optimum speed. Line 5400 calls this test-a-character routine, shown in Fig. 7-3, which will be discussed shortly.

When the program returns from testing, at least one of several flags will have been set to "on" (-1):

DUN Indicates that the typist has pressed RETURN or ESCape, and is done with the line.

BKS Indicates that DEL or an arrow key was pressed, which will cause the program to invoke a special backspace routine.

IN Tells whether the typed character is to be in the IN$ variable. Backspaces, RETURNs, and ESCapes won't be placed in the variable, due to the way we've set things up. You could use this same variable to keep other keystrokes out of the IN$ entry.

As you can see, these few flags exercise a lot of control over how the final entry comes out. The routine that sets them can combine these flags into many different arrangements.

Screening Character Entry

While the previous routine is where the action takes place, the routine we're about to discuss is probably the most interesting; it allows you to control, character by character, exactly what keystrokes make it into IN$, and which ones byte the dust.

As you can see in Fig. 7-4, the routine starts out by assuming that DUN = 0 (we are not DUN yet), that IN = -1 (the character should be placed in the entry string), and that this is not a backspace character. It also assumes that the key press was not the ESCape key. Unless something happens in the remainder of this test-a-character routine, these default settings will remain. But a lot can happen in this routine.

First, lines 5440 and 5460 test to see if RETURN or ESCape was pressed. If either key is encountered, the routine sets DUN to -1 (you're done with entry), and IN to 0 (you don't want to include either RETURN or ESCape in the string).

Line 5440 also calls a routine that clears the entry field to the right (from the current line), thereby removing any extra characters that may remain on the line.

Notice that a special ESC flag is included as part of line 5460. ESCape (or any other special key) can serve as an *escape hatch* for times when users

want to get out of the current routine. This flag will remain set when the entry routine returns to other parts of the program. These other sections of the program can test for ESC, and take appropriate action if it is set to −1.

Backspaces. You've probably noticed the lines getting longer and longer. The next one's a real thicket of decision making. It tests for all cursor characters and other keystrokes that might confuse the routine. The symbols and their ASCII codes are not readily apparent from this line, so here they are:

(CH>144 AND CH<149)	Tests for down arrow, RVS OFF, CLR HOME, and INS.
CH=157	Tests for the cursor-left key.
CH=95	Tests for the leftward-arrow key at the upper left side of the keyboard.
(CH<32)	Tests for any invisible *control* characters.

If any of these keys was the one pressed, BKS is set to on. In effect, this line converts all of these keystrokes to backspaces. Because you wouldn't want a backspace character to print, IN is set to zero. Since no further tests are necessary before accepting the next character, this line goes to 5530—the end of the subroutine.

Converting Graphics Characters. Line 5500 performs the functions of converting shifted keyboard characters, which would normally appear as graphic symbols, into normal letters. It does this by shifting down the ASCII code by a value of 128, which is the gap separating letters from graphic characters. Think of it as singing the same song in a lower octave.

If your program worked in the lowercase/uppercase mode, a similar routine could be used to lock out lowercase characters by shifting them upward by 128. You'd first have to detect whether the character was alphabetic lowercase in order to ensure you didn't step on any other tests in the routine. Thus:

```
(CH>64 AND CH<91)
```

would replace the CH>96 test in the current line.

Testing for Numbers. Back in the Setup Variables section we mentioned a variable named NMBR, which is set to −1 when the entry should accept numbers only. NMBR comes into play at line 5505.

The test on this line takes advantage again of

```
5400 :
5410 dun=0:in=-1:bks=0:esc=0
5415 :
5420 rem                    test character
5430 ln=len(in$): rem cur len
5440 if ch=13 then dun=-1:in=0: gosub 5800: rem cr
5460 if ch=27 then dun=-1:in=0:esc=-1:rem esc
5480 if (ch>144 and ch<149) or ch=157 or ch=95 or (ch<32)then bks=-1:in
     =0:goto 5530
5500 if ch>96 then ch=ch-128:ch$=chr$(ch): goto 5520
5505 if nmbr and (ch>57 or ch<45) then play bleep$ :in=0:goto 5530
5520 if ln=>le then play bleep$:in=0
5530 return
```

Fig. 7-4. A test character routine.

ASCII's grouping related characters together. Since the numbers 0-9 are located in the ASCII table between 45 and 57, the routine assumes that anything below or above that range is not a number. A bleep is sounded. IN is set to zero, since whatever was typed shouldn't be placed in the entry string or displayed on the screen. Then the line branches to 5520.

If your eyes are sharp, you've probably noticed that the GOTO 5520 here is unnecessary; 5520 is the very next line. The GOTO branch is included here as a safety feature, to ensure that additional lines that are later added won't gum up the way this routine works.

Testing for Length. It's always best to catch an error before it happens. Line 5520 does just that. If the current entry length (LN) is about to exceed the allowed length of entry (LE), the BLEEP is played, and IN is set to zero.

Customizing a Screen Test. Sometimes it's not enough just to screen entry length or to test for numbers versus letters. Sometimes you need to block entry of all but a small group of characters.

Take a budget program, where an expense might be labeled with a one-letter code to designate daily, weekly, biweekly or monthly. The prompt on the screen would look like this:

```
PLEASE ENTER CODE (D/W/B/M): -
```

The program user would simply type the code and press RETURN.

Naturally, you could test the user's input after the fact to determine if one of the four valid codes had been entered. But wouldn't it be more elegant to simply not allow entry of incorrect characters to begin with? Especially if you could do it with only one short line of code?

The secret command is our old friend INSTR. Here's what such a screening line would look like at line 5510:

```
5510 IF INSTR("DWBM",CH$) <1 THEN
     PLAY BLEEP$: IN=0
```

This line simply looks for one of the four characters. If one of the four is not stored in CH$, the line bleeps a protest and sets IN to zero, barring the character from entry into IN$.

To make the line more generic, so it could be changed depending on the characters needing to be screened, a string variable should be substituted for DWBM:

```
5510 IF INSTR(ALLOWED$,CH$) <1
     THEN PLAY BLEEP$: IN=0
```

Another variable should be included, so that the line will be executed only if this particular type of test is desired:

```
5510 IF ALLOW THEN IF
     INSTR$(ALLOWED$,CH$) <1 THEN
     PLAY BLEEP$: IN
```

The ALLOW flag should be reset to zero at the end of entry, preferably just before the RETURN statement goes back to the main program, as shown below:

```
5395 ALLOW=0: RETURN
```

The INSTR operations we've been discussing are incredibly fast. I've placed the entire alphabet into ALLOWED$ and still witnessed no decline in entry speed. Of course, if you use this routine, don't forget to activate it with ALLOW, and don't forget to place the proper characters in ALLOWED$. Otherwise, the routine will be searching for blanks, and the program won't allow entry of anything at all.

The Backspace and Other Routines

We've taken these pages to cover the peripheral routines because they're all short and simple. The clear-to-end-of-field routine at line 5805 in Fig. 7-5 determines the remaining empty space in a field (LE-LN), and prints that number of spaces—obliterating anything to the right of the cursor (to the border of the entry, anyhow).

The clear-buffer routine at 5905 uses an eight-

```
5805 rem          clear to end of field
5810 :  char ,h,v,case$+in$+left$(space$,le-ln)+reset$
5820 :
5830 return
5900 :
5905 rem                    clear buffer
5910 forzz=1 to 8:get ch$:next
5920 return
5950 :
5960 :rem                  backspace rtn
5970 if ln=0 then tempo 255: play"s v1o5 c v2o6 c v3o4c":goto 5995
5980 in$=left$(in$,ln-1)
5995 return
```

Fig. 7-5. A routine to clear text from a field.

step loop to GET any characters that might have been typed. The program doesn't do anything with them, it just GETs them. If nothing was typed, that's OK too; the GET command doesn't require that a character be typed. It only checks to see if there is one, and grabs it up if it exists.

The backspace routine, starting at line 5970, is a little more complex because it deals in illusion. The first illusion it proffers is that there is a wall at the far left of the entry line. When a backspace has been pressed, and the length (LN) of IN$ is zero, this PLAY routine will produce a banging noise to indicate that the cursor has hit this wall.

The second trick is at line 5980, where the rightmost character is lopped off of IN$. This is done by using LEFT$ to include all characters but the last one in a newly defined IN$.

Time Travel

Humans must be awfully boring to microcomputers. After all, while a human is trying to decide on the next key to press, the Commodore 128 could be performing dozens of other operations. Yet, using GETKEY, it just sits, waiting patiently for its master to mash the next key.

It needn't be so. With a little forethought, you can have your computer off doing other things while you're deciding which keys to press. The command that will make it all happen is GET, discussed at the beginning of this chapter and used in the clear-keyboard-buffer routine.

To refresh your memory, GET is like GETKEY, except that it doesn't wait. If there's a keystroke sitting there in the keyboard buffer, GET will grab hold of it and place it in the appointed variable. If there isn't a character there, the variable will be empty.

You can see the only way to work it is to keep testing for a keystroke until one is found:

```
5140 CH$=" ":DO WHILE CH$=" ":GET
     CH$:LOOP
```

It's exactly what GETKEY does automatically. In fact, a line like the one above doesn't serve much purpose; you could do the same thing with GETKEY much more easily.

But what if we introduce a new factor—a subroutine?

```
5140 CH$=" ":DO WHILE CH$=" ":GET
     CH$:GOSUB 5700:LOOP
```

Now, every time the line tests for a keystroke, it also goes down to a subroutine to perform another operation. The entry routine is in effect doing double duty—waiting for a keystroke while also performing another operation.

In this case, the other operation displays the time, which is done through the routine shown in

```
5700 ::::::::::::::::::::::::::::::::
5710 rem                    dislay time
5720 char ,0,24,left$(ti$,2)+":"+mid$(ti$,3,2)+":"+right$(ti$,2)
5730 return
```

Fig. 7-6. Displaying the time.

Fig. 7-6. This routine takes full advantage of the C-128's built-in clock. With a little help from BASIC string handling commands, line 5720 places the current time on the lower right corner of the screen.

The visual effect is of a clock ticking away the seconds, while the computer awaits keyboard entry. It's all so fast that even when you're typing at full speed, the time piece remains undisturbed . . . as steady as Big Ben.

Setting the Time. To use this routine properly, you should set the time at the beginning of your program. The time variable TI$ is set in the following manner, according to a 24-hour clock:

8:30:00 am	TI$ = "083000"
12:00:20 pm	TI$ = "120020"
2:40:10 pm	TI$ = "024010"

The *leading zeroes* in these equations are mandatory. If you don't include all six digits in the time command, the computer will slap you with an electronic traffic ticket: "ILLEGAL QUANTITY ERROR."

Whistle While You Wait

Of course, displaying the time isn't the only option. What if you could play music? Figure 7-7 shows a routine that plays a note from a MUSIC$ variable each time it is called. Simply change the GOSUB 5700 to GOSUB 5600 at line 5140, and you'll have a ready-made reprise of Shave and a Haircut, ad nauseum. The music variable must be set up in the 60000's like this:

```
60093 MUSIC$ = "WFRR.CR.CR.DR.
      CRRR.ERRFRRRR":LMT=LEN
      (MUSIC$)
```

Again, don't worry if you're not yet up on the PLAY command. It's covered in another chapter. The point of these exercises is that you can do almost any simple task while the computer is awaiting keyboard entry.

COMPLETE LISTINGS

Because the entry routines discussed in this chapter include so many scattered subroutines, they are reproduced in full at the end of this chapter, in Figs. 7-8 and 7-9. If you've had trouble following any portion of these routines, you can probably get a better understanding of them now by examining these listings.

Both of these routines include dummy main processing routines which enable them to be easily tested.

USING THE ENTRY ROUTINES WITH ARRAYS

The entry routines discussed in the chapter are versatile enough to be used in almost any type of program you design. Often you'll want to use this routine in conjunction with arrays. It's just a matter of setting up the proper beginning variables, calling the entry routine, and finally, assigning IN$—the variable returned from the entry routine:

```
5600 ctr=ctr+1
5605 tempo 100
5610 play "o2"+mid$(music$,ctr,1)
5620 if ctr=lmt then ctr=0
5625 tempo 255
5630 return
```

Fig. 7-7. Playing while waiting.

```
5 rem nmbr=-1
10 h=0
20 v=10
25 le=15
30 scnclr
35 char,h,v,"xxxxxxxxxxxxxxxxxx"
40 gosub 60000
50 gosub 5000
60 rem printin$
65 rem if esc then print "escape was pressed"
70 end
5000 ::::::::::::::::::::::::::::::::::
5020 rem                    input routine
5040 :
5050 tempo 255
5060 gosub 5900          : rem clr buffer
5080 in$="":ch$="":in=0:dun=0
5085 :
5090 if le>240 then le=240
5100 :
5120 do until dun
5130 :  char ,h,v,case$+in$+csr$+reset$
5140 :  getkey ch$
5160 :  ch=asc(ch$)
5180 :  gosub 5400       : rem   test ch
5190 :  if dun then exit
5200 :
5210 :  if bks then bks=0:gosub 5950
5300 :  if in then in$=in$+ch$
5390 loop
5392 :  char ,h,v,case$+in$+" "+reset$
5395 return
5400 :
5410 dun=0:in=-1:bks=0:esc=0
5415 :
5420 rem                    test character
5430 ln=len(in$): rem cur len
5440 if ch=13 then dun=-1:in=0: gosub 5800: rem cr
5460 if ch=27 then dun=-1:in=0:esc=-1:rem esc
5480 if (ch>144 and ch<149) or ch=157 or ch=95 or (ch<32) then bks=
     -1:in=0:goto 5530
5500 if ch>96 then ch=ch-128:ch$=chr$(ch): goto 5520
5505 if nmbr and (ch>57 or ch<45) then play bleep$ :in=0:goto 5530
5520 if ln=>le then play bleep$:in=0
5530 return
5800 ::::::::::::::::::::::::::::::::::
5805 rem          clear to end of field
```

Fig. 7-8. An entire input listing.

```
5810 :  char ,h,v,case$+in$+left$(space$,le-ln)+reset$
5820 :
5830 return
5900 :
5905 rem                    clear buffer
5910 forzz=1 to 8:get ch$:next
5920 return
5950 :
5960 :rem                 backspace rtn
5970 if ln=0 then tempo 255: play"s v1o5 c v2o6 c v3o4c":goto 5995
5980 in$=left$(in$,ln-1)
5995 return
60000 :
60094 bleep$="s v1o5 g v2o6 g v304 g"
60095 space$=""
60096 space$=space$+space$+space$
60097 csr$=" "
60098 case$=chr$(142)
60099 reset$=chr$(27)+"c"
61000 return
```

```
5 rem nmbr=-1
10 h=0
20 v=10
25 le=15
30 scnclr
40 gosub 60000
50 gosub 5000
60 rem printin$
65 rem if esc then print "escape was pressed
70 end
5000 ::::::::::::::::::::::::::::::::::
5020 rem                    input routine
5040 :
5050 tempo 255
5060 gosub 5900          : rem clr buffer
5080 in$="":ch$="":in=0:dun=0
5085 :
5090 if le>240 then le=240
5100 :
5120 do until dun
5130 :  char ,h,v,case$+in$+csr$+reset$
5140 :  ch$="":do while ch$="":get ch$:gosub 5700:loop
5160 :  ch=asc(ch$)
```

Fig. 7-9. An entire listing for "Time Input." Logic for music is also included.

```
5180 :  gosub 5400       : rem   test ch
5190 :  if dun then exit
5200 :
5210 :  if bks then bks=0:gosub 5950
5300 :  if in then in$=in$+ch$
5390 loop
5392 :  char ,h,v,case$+in$+" "+reset$
5395 return
5400 :
5410 dun=0:in=-1:bks=0:esc=0
5415 :
5420 rem                    test character
5430 ln=len(in$): rem cur len
5440 if ch=13 then dun=-1:in=0: gosub 5800: rem cr
5460 if ch=27 then dun=-1:in=0:esc=-1:rem esc
5480 if (ch>144 and ch<149) or ch=157 or ch=95 or (ch<32) then
     bks=-1:in=0:goto 5530
5500 if ch>96 then ch=ch-128:ch$=chr$(ch): goto 5520
5505 if nmbr and (ch>57 or ch<45) then play bleep$ :in=0:goto 5530
5520 if ln=>le then play bleep$:in=0
5530 return
5600 ctr=ctr+1
5605 tempo 100
5610 play "o2"+mid$(music$,ctr,1)
5620 if ctr=lmt then ctr=0
5625 tempo 255
5630 return
5700 ::::::::::::::::::::::::::::::::::
5710 rem                       dislay time
5720 char ,0,24,left$(ti$,2)+":"+mid$(ti$,3,2)+":"+right$(ti$,2)
5730 return
5800 ::::::::::::::::::::::::::::::::::
5805 rem            clear to end of field
5810 :  char ,h,v,case$+in$+left$(space$,le-ln)+reset$
5820 :
5830 return
5900 :
5905 rem                      clear buffer
5910 forzz=1 to 8:get ch$:next
5920 return
5950 :
5960 :rem                  backspace rtn
5970 if ln=0 then tempo 255: play"s v1o5 c v2o6 c v3o4c":goto 5995
5980 in$=left$(in$,ln-1)
5995 return
60000 :
60093 music$="wfrr.cr.cr.dr.crrr.errfrrrr":lmt=len(music$)
60094 bleep$="s v1o5 g v2o6 g v304 g"
```

```
60095 space$="
60096 space$=space$+space$+space$
60097 csr$=" "
60098 case$=chr$(142)
60099 reset$=chr$(27)+"c"
61000 return
```

```
4000  FOR ITEM = 1 TO FIELDS
4010  : H=H(ITEM) : V=V(ITEM)
4020  : LE=LE(ITEM)
4030  : GOSUB 5000 :REM ENTRY
        ROUTINE
4040  : A$(ITEM) = IN$
4050    NEXT
```

The variables H(ITEM), V(ITEM), and LE(ITEM) are used to store the horizontal and vertical positions, and length of each item to be entered. These values are assigned to H, V, and LE—the variables used by the entry routine itself—at lines 4010 and 4020.

Line 4030 calls the input subroutine, and line 4040 assigns the fruit of this routine (IN$) to the A$(ITEM) variable. In effect, we've used a few variable assignments and a GOSUB to replace a regular input routine that would have looked like this:

```
4000  FOR ITEM = 1 TO FIELDS
4010  :  CHAR ,H(ITEM),V(ITEM)
4020  :  LN=LE+1:DO UNTIL LN <= LE
4025  :  INPUT A$(ITEM)
4030  :  LN=LEN(A$(ITEM)
4040  :  LOOP
4050  NEXT
```

As you can see, the "simple" array input routine is even more complicated than our souped-up version—even though this "regular" routine doesn't do nearly as much.

USING THE FUNCTION AND HELP KEYS

When the Commodore 128 is powered up, the system's function keys are ready for action—as long as your actions involve programming. But press a function key from within a BASIC program, and you'll see GRAPHIC, DLOAD", or some other command. The keys aren't initialized or reassigned unless you specifically do so.

The Commodore 128 doesn't have the ON KEY . . . GOSUB command common to many business computers. Fortunately, these function keys can be assigned special codes, which can then be trapped during entry routines.

The command to assign a code to the F1 key looks like this:

```
KEY 1, "HELLO"
```

Because there's no + CHR$(13) at the end of this assignment, the typist could add letters to the end of HELLO after F1 was pressed. If CHR$(13) had been added to the assignment, pressing F1 would be the same as typing HELLO and pressing RETURN.

Normally, you'll probably want to assign function keys as special codes that will be intercepted in the entry routine, much as we did with ESCape. It's best to use the codes between CHR$(171) and CHR$(183), since they are produced by pressing the C= key and are seldom pressed accidentally:

```
KEY 1, CHR$(176)  :REM SAME AS C= + A
```

If you wish to "blank out" the default value of a key, simply assign it a value of empty quotes (a null). You can use a loop to disable all of the F-keys:

```
FOR X=1 TO 8: KEY X, " ": NEXT
```

Defining the Previous Entry

Because function keys may be assigned through string variables (in addition to string constants such as "HELLO"), you can continually update F-keys within a running program. For example, you might wish to store the last input value in function key one, which could be later pressed to repeat a previous entry. To store each new IN$ variable in F1, simply include a line such as this at the end of your entry routine

```
KEY 1, IN$ : RETURN
```

The next time F1 is pressed, the IN$ value from the previous entry will appear.

Reassigning the Help Key

BASIC 7.0 isn't so helpful when it comes to reassigning the HELP key. Yet HELP is one of the first keys you may consider scanning for in your entry routine. After all, your programs will seem a lot more professional if a user can always count on an answer after pressing HELP.

Here's a brief routine that will reassign this key, using a few tricks of BASIC:

```
59000 HLP$="OOPS"+CHR$(13)
59005 HLP$=LEFT$(HLP$,5)+"     " :
      REM FIVE SPACES
59010 FOR ZZ=1 TO 5
59020 : POKE
      4118+ZZ,ASC(MID$(HLP$,ZZ,1))
59030 NEXT
59040 RETURN
```

Remember that the value you assign to HELP cannot be longer than four characters, plus a carriage return, if one is used. If you do not plan to use all four characters, this routine *pads* the remaining characters with spaces. If you do not use all five characters, this routine pads the remainder of the HLP$ string with spaces. Naturally; the carriage return is considered a character.

Once this routine is installed in your program, you can test for help using the same procedure explained for ESCape in the section on screening character entry. If you want the program to stay where it is, you can have the HELP line call a subroutine instead of returning from entry (as ESCape does).

Chapter 8

Sorting

Used to their fullest, sorting routines can do a lot more for you than simply placing an array of items in alphabetical order. Sorts can group items together, help your programs count related information, and set up data to be searched at lightening speed.

WHAT'S A SORT

Volumes and volumes have been written about how to sort in BASIC. Yet the heart of any BASIC sort routine comes down to three simple statements:

```
T$ = ITEM$(1)
ITEM$(1) = ITEM$(2)
ITEM$(2) = T$
```

It's a simple swap, and that's what sorting's all about: swapping items until they're in the right order.

The real complications in sorting come from trying to do the sort with the least amount of swapping possible. Figure 8-1, a very slow sort, is easy to follow.

The meat of the program is the sorting routine beginning at line 8000. The sort starts at line 8020, by setting up a loop variable (LEAD) that will point to each item, from the first to the next-to-the-last item. The second variable, FOL, will always be one greater than LEAD. Using these two variables, we can compare adjacent items in the list, and swap them if they are not in order.

Line 8040 does the test. If the LEAD item is greater than the second item, the program performs a swap. Otherwise, the program continues on. The SWPD flag in the routine prevents the outside loop from being repeated if no swaps occurred (indicating that all items were in order during the previous pass of the inside loop).

This type of sort is called a *bubble sort*, because the lesser items gradually float to the top of this list like bubbles in a glass of ginger ale.

Running the bubble sort program in Fig. 8-2 will clearly illustrate how this type of sort works. This program highlights each data item as it is

```
Ø :           goto 1Ø
2 : simple sorting program
3 :
1Ø dim item$(5Ø)
2Ø scnclr Ø
22 limit=1Ø : rem screen display limit
25 item=1:  rem set first item
26 :
3Ø gosub 4ØØØ :  rem get entry
4Ø gosub 8ØØØ:   rem sort
5Ø gosub 2ØØØ :  rem display
55 :
6Ø :
7Ø end
8Ø :
2ØØØ : rem display
2Ø1Ø :
2Ø15 scnclr
2Ø2Ø for i=1 to item
2Ø25 :
2Ø3Ø :   print item$(i)
2Ø4Ø :   ctr=ctr+1
2Ø45 :
2Ø5Ø :   if ctr=limit then begin
2Ø6Ø :      ctr=1
2Ø65 :      char ,1Ø,23,"press a key"
2Ø7Ø :      getkey a$
2Ø75 :      scnclr
2Ø8Ø :   bend
2Ø85 :
2Ø9Ø next
21ØØ return
4ØØØ : rem input routine
4Ø1Ø :
4Ø15 print"enter items. return when done."
4Ø16 print
4Ø2Ø do
4Ø3Ø :  print "item #: ";item;
4Ø4Ø :  input item$(item)
4Ø5Ø :  if item=5Ø or item$(item)="" then item=item-1:exit
4Ø6Ø :  item=item+1
4Ø7Ø loop
4Ø75 :
4Ø8Ø return
4Ø9Ø :
8ØØØ : rem sort routine
```

Fig. 8-1. A typical bubble sort program.

```
8005 :
8006 scnclr:char ,15,10,"now sorting"
8007 :
8010 :
8020 for lead=1 to item-1
8022 :   swpd=0
8030 :   for fol=lead+1 to item
8035 :
8040 :       if item$(lead)=>item$(fol) then begin
8042 :          t$=item$(lead)
8044 :          item$(lead)=item$(fol)
8046 :          item$(fol)=t$
8047 :          swpd=1
8048 :       bend
8050 :   next
8052 :   if swpd=0 then 8070
8055 :
8060 next
8070 return
```

```
0 :            goto 5
2 : bubble sort
3 :
4 :
5 scnclr
10 gosub 60000
20 gosub 8000   :rem sort & display
30 lead=0:fol=0:gosub 8800 :rem final display
40 end
8000 ::::::::::::::::::::::::::::::::
8010 rem                  sort routine
8020 for lead=1 to last-1
8030 for fol = lead+1 to last
8040 if a$(lead) <= a$(fol) then 8050
8045 t$=a$(lead):a$(lead)=a$(fol):a$(fol)=t$:dun=0
8048 gosub 8800
8050 next
8060 next
8070 return
8800 ::::::::::::::::::::::::::::::::
8810 rem            display sorted list
8820 :
8825 print
```

Fig. 8-2. How a bubble sort works. When run, this program shows data items as they are swapped, giving you a window into the bubble sort.

```
8830 for rec=1 to last
8835 if rec=fol or rec=lead then rv=1:else rv=0
8836 if rec=lead then play "qv1c v2e"
8837 if rec=fol then play "qv1e v2g"
8840 : char ,0,rec,left$(a$(rec)+space$,15),rv
8850 next
8860 return
60000 rem     :load up list:
60005 space$="           "
60010 a$(1)="chili dog"
60020 a$(2)="hamburger"
60040 a$(3)="pizza"
60060 a$(4)="paella"
60080 a$(5)="filet mignon"
60082 a$(6)="submarine"
60084 a$(7)="dog food"
60086 a$(8)="scrambled eggs"
60088 a$(9)="eggs benedict"
60089 a$(10)="country salad"
60090 last=10:  rem  there are 10 items
60100 return
```

swapped. When the program gets to the end, all items will be in order.

Both of these sorts are easy to understand. More complicated sorts, such as quicksort, shown in Fig. 8-3, are harder to follow, but are much more efficient because they perform fewer swaps to get the data in order.

The quicksort routine can be used in place of the slower bubble sort routine shown in Fig. 8-1.

SORTING TWO-DIMENSIONAL ARRAYS

The only thing that changes in the sort of a bi-dimensional array is the test and the swap. Let's say you wanted to sort an array ITEM$-(REC,FIELD) that's broken down like this:

ITEM$(REC,1)	refers to last name
ITEM$(REC,2)	refers to city
ITEM$(REC,3)	refers to zip code

After deciding which field would drive the sort (do you want to sort on last name? city? zip code?), you would simply *hard code* this number into the test, like so:

```
8040:  IF ITEM$(LEAD,2)=
       >ITEM$(FOL,2) THEN BEGIN
```

LEAD and FOL still refer to records in the array. The only thing that's changed so far is that you've added another dimension. You're sorting on the value in the second field (city) of each record.

The next step is to fix the swap routine. If there are now three elements in each record, each must be swapped:

```
FOR CTR=1 TO 3
   T$=ITEM$(LEAD,CTR)
   ITEM$(LEAD,CTR)=ITEM$(FOL,CTR)
   ITEM$(FOL,CTR)=T$
NEXT
```

If there is a substantial number of fields in each

```
Ø :           goto 1Ø
2 : quicksort program
3 :
1Ø dim item$(5Ø),stack(2Ø,2)
15 true=1:false=Ø
17 :
2Ø scnclr Ø
22 limit=1Ø : rem screen display limit
25 item=1:  rem set first item
26 :
3Ø gosub 4ØØØ :  rem get entry
4Ø gosub 8ØØØ:   rem sort
5Ø gosub 2ØØØ :  rem display
55 :
6Ø :
7Ø end
8Ø :
2ØØØ : rem display
2Ø1Ø :
2Ø15 scnclr
2Ø2Ø for i=1 to item
2Ø25 :
2Ø3Ø :   print item$(i)
2Ø4Ø :   ctr=ctr+1
2Ø45 :
2Ø5Ø :   if ctr=limit then begin
2Ø6Ø :      ctr=1
2Ø65 :      char ,1Ø,23,"press a key"
2Ø7Ø :      getkey a$
2Ø75 :      scnclr
2Ø8Ø :   bend
2Ø85 :
2Ø9Ø next
21ØØ return
4ØØØ : rem input routine
4Ø1Ø :
4Ø15 print"enter items. return when done."
4Ø16 print
4Ø2Ø do
4Ø3Ø :  print "item #: ";item;
4Ø4Ø :  input item$(item)
4Ø5Ø :  if item=5Ø or item$(item)="" then item=item-1:exit
4Ø6Ø :  item=item+1
4Ø7Ø loop
4Ø75 :
4Ø8Ø return
```

Fig. 8-3. A quicksort program. Quicksort is always the fastest for big lists, and is often faster even when sorting small lists.

```
4090 :
8000 : rem sort routine
8005 :
8006 scnclr:char ,15,10,"now sorting"
8007 :
8010 :
8020 pntr=1:stack(1,1)=1:stack(1,2)=item
8030 do until pntr=0
8040 si=stack(pntr,1)
8050 sj=stack(pntr,2)
8060 pntr=pntr-1
8070 do while si<sj
8080 lead=si
8090 fol=sj
8100 test=true
8110 do while lead<fol
8120 if item$(lead)>item$(fol) then begin
8130 :  t$=item$(lead)
8140 :  item$(lead)=item$(fol)
8150 :  item$(fol)=t$
8160 :  test=-test
8170 bend
8180 if test=true then lead=lead+1: else fol=fol-1
8190 loop
8200 if lead+1<sj then begin
8210 :  pntr=pntr+1
8220 :  stack(pntr,1)=lead+1
8230 :  stack(pntr,2)=sj
8240 bend
8250 sj=lead-1
8260 loop
8270 loop
8280 return
```

each record, this kind of swapping can really slow down a sort, so a lot of programs concentrate only on the field being sorted, by assigning it into its own single-dimension array, and setting up a record pointer that is sorted in parallel. Assuming the single-dimension array were set up as A$(ITEM), the test and swap would look like this:

```
If A$(LEAD)= >A$(FOL) THEN BEGIN
   T$=A$(LEAD)
   A$(LEAD)=A$(FOL)
   A$(FOL)=T
:
   T=RC(LEAD)
   RC(LEAD)=RC(FOL)
   RC(FOL)=T
:
BEND
```

The RC array would then serve as a sorted *pointer* to records. Even though the record themselves are not in sorted order, they can be printed

that way by referring to them with RC:

```
FOR I=1 TO ITEM          : REM RECORDS
   FOR J=1 TO 3          : REM FIELDS
      PRINT ITEM$(RC(I),J)
   NEXT
   PRINT
                         : REM SPACE
                           BETWEEN
                           RECORDS
NEXT
```

The only real drawback to this approach is that the records are never sorted. If you want to write them back to the disk in sorted order, you have to include the RC variable in your write-to-disk routine.

SORTING NUMBERS

BASIC does a fine job of sorting numbers that are stored in a numeric array. It does splendidly with words or phrases within a string array. But BASIC falls short when it comes to sorting numbers within strings. The reason is that your computer treats numbers within strings as though they should be in alphabetical order:

```
1
1000
2
22
3
```

This approach makes perfect sense to the computer, because it deals with strings from the left to right. Since one comes before two, which comes before three, the computer places 1, 1000, 2, 22, and 3 in that order.

There are a couple of ways around this problem. The first is obvious: store and sort your numbers in arrays. If you can't do that, you can *left pad* the number with zeroes during entry (0001, 0002, etc), or left pad strings that start with numbers after entry:

```
IF VAL(A$(ITEM)>0 THEN BEGIN
   L=LEN(A$(ITEM))
   FOR ZZ=1 TO 10-L
      A$(ITEM=" "+A$(ITEM)
   NEXT
BEND
```

The routine above assumes that the maximum length of the field is 10. You can change the value 10 to whatever number is appropriate.

A FINAL NOTE ON SORTS

Try not to look exclusively at sorts as routines to alphabetize phone lists or album collections. They're really much more. With sorts, you can group information in a variety of ways. For example, a simple subroutine can be added to a display routine that will provide counts and subtotals on related records:

```
ALABAMA  MONT.              $100
ALABAMA  SELMA              $ 50
-----------------------------------------
TOTAL FOR ALABAMA:          $150
-----------------------------------------
FLORIDA  JAX                $ 90
FLORIDA  TAMPA              $ 40
. . .
```

During the print routine, the program simply remembers the last item and compares the current one to it. If there's a match the program tallies up a new dollar amount. If there is no match, the program prints a total and moves on.

Another application of sorts is the binary search, which we touched on in Chapter 4. If your data is kept sorted, you can use these binary searches and greatly speed access to your information.

Finally, sorts can help you group related items together. Sorting records by different fields lets you look at your data in new ways, and often points out patterns you never would have otherwise noticed. The sort routines in this chapter should become part of your permanent program library.

Chapter 9

Professional Program Design: Appearance

When you think about it, there really isn't very much to separate a professionally designed and marketed program from the "free" public domain software you come across from users groups and friends. In fact, many "home brew" software programs are at least as well-designed internally as those sold in computer stores. The difference is that home-brew programs often look rough around the edges; the screen displays are barren; there may not be sufficient error trapping; and the program may not screen incorrect keystrokes or provide enough help when things go wrong. These problems detract from the appearance and usefulness of a program. And unfortunately many folks judge a program by what it looks like—not how well it functions internally.

CREATING ENTRY FORMS

One of the most important areas to attack is information entry, because this is where users spend most of their time and where you want to make them most comfortable—and trap as many errors as possible.

We've already covered how to trap certain keys and define the length of items. Now, let's see how *records* and other data can be entered more professionally.

Many commercial programmers like to design their entry screens as *forms*. Users are supposed to draw a mental connection between screen entry and filling in business forms or blanks on an index card. It works, and it works well.

This points to the cardinal rule in program design: Whenever possible, design your program so that it parallels some similar operation in the real world. We're all most comfortable with the things we know.

Not only does it make it easier to enter information; as shown in Figs. 9-1 and 9-2 this approach also enables you to spread out the items on the screen, so more items will fit, and the screen will appear less cluttered.

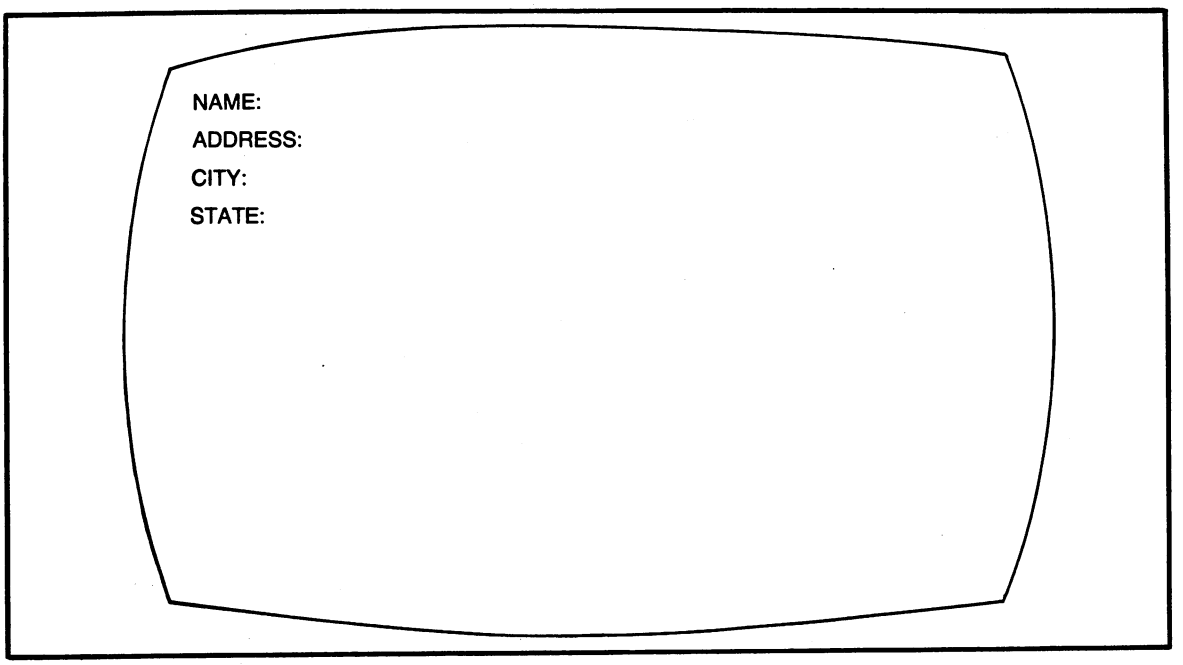


Fig. 9-1. An entry screen without an input form.

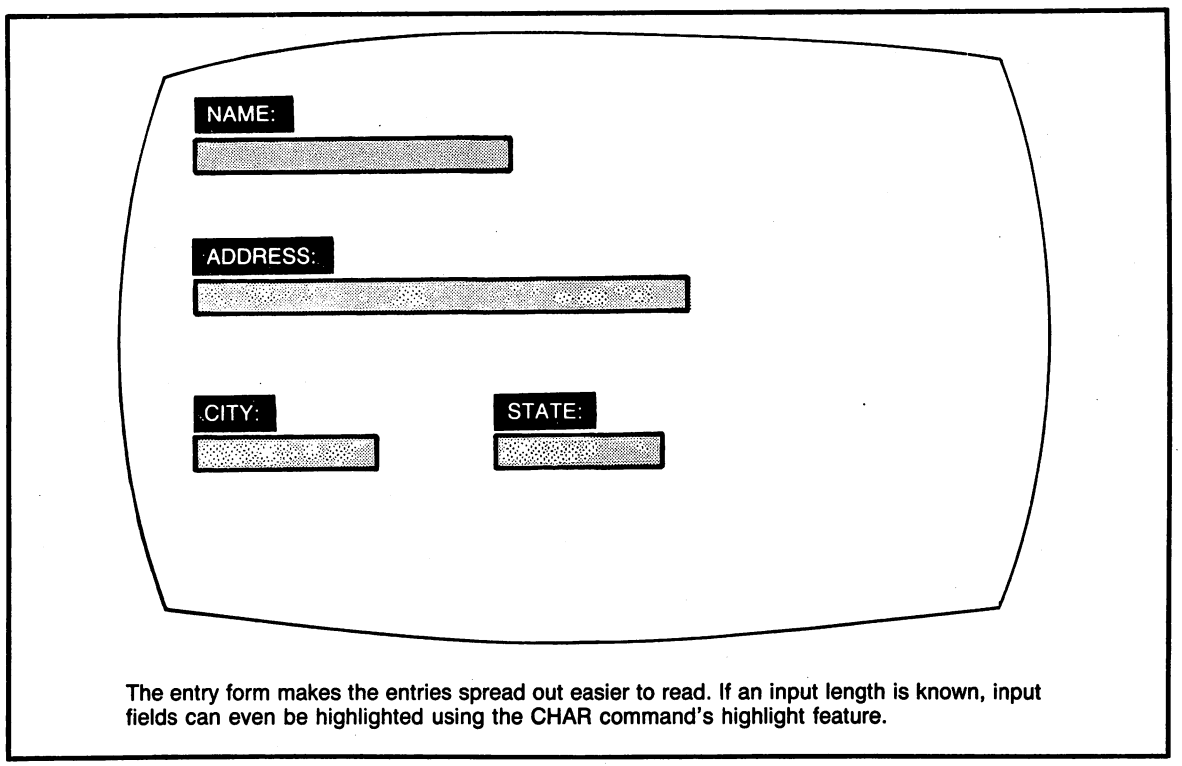


Fig. 9-2. An entry screen with an input form.

There are several ways of locating items on the screen in Commodore 128 BASIC. By far the easiest to understand and most versatile, however, is the CHAR command. It's short for *character*, but it's pronounced "char" as in charcoal. The CHAR command does the following:

- Defines on which graphic screen the character will be drawn.
- Defines column and row coordinates.
- Prints the text and character graphics you specify.
- Determines whether the characters will be displayed in reverse or normal form.

The format of the CHAR command is a little difficult to follow at first, but you'll soon be comfortable with it.

```
CHAR ,10,20,"HELLO",1
```

The first comma after CHAR tells the computer to use the *default* (skip, if you will) optional *color source*. Next comes the graphic screen, the column you wish to print at, and the row you wish it to start at, followed by the string you wish to appear, and whether reverse is on or off. Figure 9-3 shows some examples of the CHAR command.

CHAR has several advantages over the PRINT command:

- It can be used on all screens, including graphics and 80-column modes.
- In the graphics and 80-column modes, it enables you to print to the very last column on a line, without advancing to the next line.
- It lets you reverse characters without worrying about the sometimes troublesome RVS character that is required if you're using the PRINT command.

CHAR can display any text that can be placed on the screen using PRINT. In some cases, however, a few tricks are required. CHAR cannot be used with the printer—it always goes directly to the screen.

Another rule to keep in mind is that CHAR always uses strings (either variables like A$ or literals like "HELLO"). A number used within CHAR must therefore be expressed as a string:

```
CHAR ,5,8,"10"
```

The above expression would print the number 10 at column five, row eight of the screen. The 10 is enclosed in quotes because it is being treated as a string. If you wish to display a numeric variable using CHAR, that variable must be converted into a string using the STR$ function.

Here are several examples, all of which print the number seven:

```
CHAR,10,1,"HELLO WORLD"

(displays HELLO WORLD at column 10, row 1)

CHAR,10,1,"HELLO WORLD",1

(displays HELLO WORLD at column 10, row 1—in reverse)

A$ = ''HELLO'':B$ = "WORLD":CHAR,10,1,A$+" "+B

(displays HELLO WORLD at column 10, row 1, using string variables)
```

Fig. 9-3. Different examples of the CHAR command.

```
10 CHAR ,0,10,"7" :REM NUMBER IN
   QUOTES
15 :
20 NUMBER=3+4
30 CHAR ,0,10,STR$(NUMBER) :REM
   USING THE STR$ COMMAND
35 :
40 CHAR ,0,10,CHR$(55) :REM DIRECT
   ASCII CODE FOR "7"
50 :
60 END
```

All the rules that apply to strings and string variables also apply to the CHAR command. For example:

```
CHAR ,10,15,LEFT$("HELLO THERE",5)
```

would print HELLO at column 10, row 15 of the screen. H,E,L,L, and O are the leftmost five characters of that string. You can even add strings together within the CHAR command, such as in this example, which prints the words "GOOD MORNING ROBERT":

```
10 A$="MORNING":NAME$=
   "ROBERT"
20 CHAR ,0,10,"GOOD "+A$+NAME
```

In fact, any of the parameters in the CHAR command can be specified as variables. This means ever-changing variables and switches in your program can determine:

- What text will be printed (as in the previous example).
- Where the text will appear on the screen.
- Whether text will appear in inverse or normal, etc.

Using Variables with CHAR

Let's say you want to put an item heading on the screen for FIRST NAME and accept entry of the name underneath this item heading. The CHAR command fits this application perfectly. Just to keep things interesting, we'll assign variables for every operation.

```
R = 0
C = 5
HEAD$ = "FIRST NAME"
RVSFLAG = 1
```

The command would look like this: CHAR ,C,R,HEAD$,RVSFLAG. You can easily see how the variables might be changed for display of different information at different places. Figure 9-4 shows this approach to the use of CHAR.

As we will see in a few pages, the ability of CHAR to use variables in this manner gives you a great deal of flexibility.

Using CHAR for Other Reasons

In the 40 or 80 column text modes, CHAR can perform another important function: it can position the cursor before PRINT or INPUT are used. You can actually use the CHAR command and PRINT together, and there are times when you'll want to. Certain programs may already be written and working perfectly in the 40- or 80-column text mode using the PRINT command. There's no need to convert all PRINT statements to CHAR statements when you simply want to relocate the text. Its ability to position the cursor for a PRINT statement is one of the unsung features of the CHAR command; it will come in quite handy when you want to locate text.

Continuing with the "heading" command just

```
10 :      c=0:  r=10: text$="hello world": rvs=1
20 : rem  col   row   text                 reverse on/off
30 :
40 char ,c,r,text$,rvs
```

Fig. 9-4. A routine using CHAR.

explained, let's say you want to accept input from the keyboard one line below the heading just printed. You can do so by simply using CHAR with an empty set of quotes, to position the cursor. Then simply accept the input. Because the cursor has been repositioned, input of the name will begin at column five, row one in this example:

```
10 CHAR ,5,1," "
20 INPUT NAME$
```

Note that the column remains the same as in the previous example; only the line number has changed. In fact, we could rewrite these two lines to take advantage of the variables set earlier in the program:

```
10 CHAR ,C,R+1," "
20 INPUT NAME
```

The statements above position the cursor one line below the NAME heading.

Note also that the default (normal) form of CHAR is for normal, not reverse, print, so it is not necessary to place a ,0 next to the empty quotes.

In addition to using this trick when rewriting existing programs, you may want to use it in routines in which information will be sent to a printer. The printer will ignore the CHAR, but will print the data that is part of PRINT. Text that is part of a CHAR will *always* appear on the screen. Text that's part of a PRINT may be redirected to the printer, a disk file, or other devices.

Now that you've seen how CHAR can be used to position characters on the screen in one example, let's see how an array of variables could be used to position such heading for five or six different items. If you're storing your information in an array, the column and row locations can be stored in separate but related arrays, as shown in lines 20 through 40 in Fig. 9-5.

Here are the steps required to use these headings properly:

- Call a subroutine to "draw" all headings at once.
- Call a subroutine to "draw" all entry underlines.
- Call a routine to input individual items.

The routine to locate these items properly is shown in lines 2400 through 2440 in Fig. 9-5.

Let's examine how the whole listing works. First, the dimensioned variable HEAD$(x) is used to store heading names. The variable R(x) is used to store the row, and C(x) is used to represent the column. So this statement:

```
HEAD$(1)="FIRST NAME":C(1)=0:R(1)=5
```

assigns "FIRST NAME" as the first item heading. It will appear in column zero (the first column), row five. As the loop counter is bumped upward with each successive pass, the values in HEAD$(x), R(x), and C(x) will change and different headings will be displayed at various places on the screen.

This is the method used by many professional filing systems in order to make data entry easier.

Now that you've seen how headings and input statements are positioned, we can look at the subject of installing a separate routine for accepting and positioning data on records to be added. Basically, this routine should do the following:

- Bump up the item counter of records in memory.
- Call a subroutine that positions and accepts input and keeps track of which field is current.
- Allow you to change the record once it has been entered.

This list is a tall order; it really calls for two separate routines: one to add to the record counter and the other to do all the real work: screen positioning, accepting input, and so on.

We can use this position and input routine in other areas, such as in a search and change routine that could easily be added to the program at a later time. You can even redesign the search routine explained in Chapter 4. This modular approach to software, where routines can be mixed and matched, is part *structured* programming.

OLD-FASHIONED MENUS

We've all seen simple menu routines that al-

```
Ø :
2 : rem routine to display headings
3 :
4 :
5 scnclr
1Ø nf=4
2Ø head$(1)=" first name ":c(1)=Ø:r(1)=1
3Ø head$(2)=" last name ":c(2)=2Ø:r(2)=1
4Ø head$(3)=" address ":c(3)=Ø:r(3)=4
45 :
46 rem above lines assign headings
5Ø :
6Ø gosub 24ØØ :rem display headings
7Ø :
75 char,Ø,22
8Ø end
24ØØ : rem   locate headings rtn
24Ø5 :
241Ø :for x=1 to nf
242Ø :    char ,c(x),r(x),head$(x),1
243Ø :next
2435 :
244Ø :return
```

Fig. 9-5. Routines to store arrays and display headings.

low you to select a number and press RETURN. Most of these menus are simple in structure, using either an IF or an ON GOSUB structure. Whatever programming approach taken, the menus generally look like the one in Fig. 9-6. They allow the user to select from a number of items and then branch to the appropriate section of the program.

Figures 9-7 and 9-8 show how IF and ON GOSUB menus are programmed.

ON . . . GOSUB

If you're not familiar with it, the ON A GOSUB function of Commodore 128 BASIC lets the program branch to a specific subroutine based on the value of A. If A = 1, the computer goes to the first subroutine in the list. If A = 3, the computer goes to the third subroutine listed, and so forth. If A is greater than the number of lines listed, the computer proceeds to the next statement. In the ON . . . GOSUB example in Fig. 9-8, the GOTO function effectively intercepts incorrectly entered numbers.

This type of menu works smoothly, and can be easily installed in a program. In fact, because its modular, you can install it as the very last thing you do, since it simply calls other subroutines. Many users like to place the menu at the bottom of the program, so that it does not clutter up the appearance of the listing. But it really doesn't matter where you put your menu. As long as it's modular, it will work anywhere.

Modularity also means that these menus can be easily replaced, so when you really want to dress up a menu, you can install a fancier version without messing up the rest of your code.

```
1. ADD A RECORD
2. EDIT A RECORD
3. SAVE FILE
4. RETRIEVE FILE
5. EXIT
YOUR CHOICE (#):_____
```

Fig. 9-6. A typical program menu.

```
Ø :            goto 5ØØØØ
2 :  menu using if
3 :
4 :
1ØØ print:print"1 selected":sleep 1:return
2ØØ print:print"2 selected":sleep 1:return
3ØØ print:print"3 selected":sleep 1:return
4ØØ print:print"4 selected":sleep 1:return
5ØØ :
6ØØ :
5ØØØØ : rem menu using if
5ØØØ5 do until esc
5ØØ1Ø :  scnclr
5ØØ15 :  char ,Ø,8
5ØØ2Ø :  print "1. add a record"
5ØØ3Ø :  print "2. delete a record"
5ØØ4Ø :  print "3. search this file"
5ØØ5Ø :  print "4. sort items"
5ØØ6Ø :  print "5. end program"
5ØØ65 :  choice=Ø    : rem initialize
5ØØ7Ø :  do while choice <1 or choice >5
5ØØ75 :     char ,Ø,14
5ØØ8Ø :     input "choice(#):";choice$
5ØØ82 :     choice=val(choice$)
5ØØ85 :  loop
5ØØ9Ø ::::::::::::::::::::::::::::::::::::
5ØØ95 :
5ØØ96 : rem only this changes from on...gosub
5ØØ97 ::::::::::::::::::::::::::::::::::::
5Ø1ØØ :  if choice=1 then gosub 1ØØ
5Ø11Ø :  if choice=2 then gosub 2ØØ
5Ø12Ø :  if choice=3 then gosub 3ØØ
5Ø13Ø :  if choice=4 then gosub 4ØØ
5Ø132 :  if choice=5 then esc=1
5Ø133 ::::::::::::::::::::::::::::::::::::
5Ø135 loop
5Ø14Ø print:print"bye!"
5Ø15Ø end
```

Fig. 9-7. A menu using IF branches.

BAR MENUS

Many commercial packages now use a menu form known as a *bar menu*. As the name implies, a highlighting bar is used to point to different options on the screen. Typically, the operator presses arrow keys or uses a mouse to move the highlighting bar to different options. When the option desired becomes "lit up," the return key or a button on the mouse can be pressed to select it. Figure 9-9 shows an example of a bar menu with a highlighted selection.

Surprisingly, the logic required for a bar menu

is not terribly much more complicated than the logic for a simple "press the item to select" type of menu.

The additional routines needed are:

1. A routine to read possible menu selections from a file or a list of data statements.
2. A routine to display the menu, highlighting the current option.
3. A routine to GET a single keystroke and to determine whether the bar should move up or down.

Usually, arrow keys are used to move the bar: A down arrow pushes the bar downward and the up arrow moves. Sometimes the right arrow and left arrow keys are also used, with right pushing the bar downward, and a left arrow keystroke moving it upward.

```
Ø :             goto 5ØØØØ
2 :  menu using on...gosub
3 :
4 :
1ØØ print:print"1 selected":sleep 1:return
2ØØ print:print"2 selected":sleep 1:return
3ØØ print:print"3 selected":sleep 1:return
4ØØ print:print"4 selected":sleep 1:return
5ØØ :
6ØØ :
5ØØØØ : rem menu using on...gosub
5ØØØ5 do until esc
5ØØ1Ø :  scnclr
5ØØ15 :  char ,Ø,8
5ØØ2Ø :  print "1. add a record"
5ØØ3Ø :  print "2. delete a record"
5ØØ4Ø :  print "3. search this file"
5ØØ5Ø :  print "4. sort items"
5ØØ6Ø :  print "5. end program"
5ØØ65 :  choice=Ø    : rem initialize
5ØØ7Ø :  do while choice <1 or choice >5
5ØØ75 :     char ,Ø,14
5ØØ8Ø :     input "choice(#):";choice$
5ØØ82 :     choice=val(choice$)
5ØØ85 :  loop
5ØØ9Ø ::::::::::::::::::::::::::::::::::::
5ØØ95 :
5ØØ96 : rem only this changes from if
5ØØ97 ::::::::::::::::::::::::::::::::::::
5Ø1ØØ on choice gosub 1ØØ,2ØØ,3ØØ,4ØØ
5Ø132 :  if choice=5 then esc=1
5Ø133 ::::::::::::::::::::::::::::::::::::
5Ø135 loop
5Ø14Ø print:print"bye!"
5Ø15Ø end
```

Fig. 9-8. A menu using ON . . . GOSUB branches.

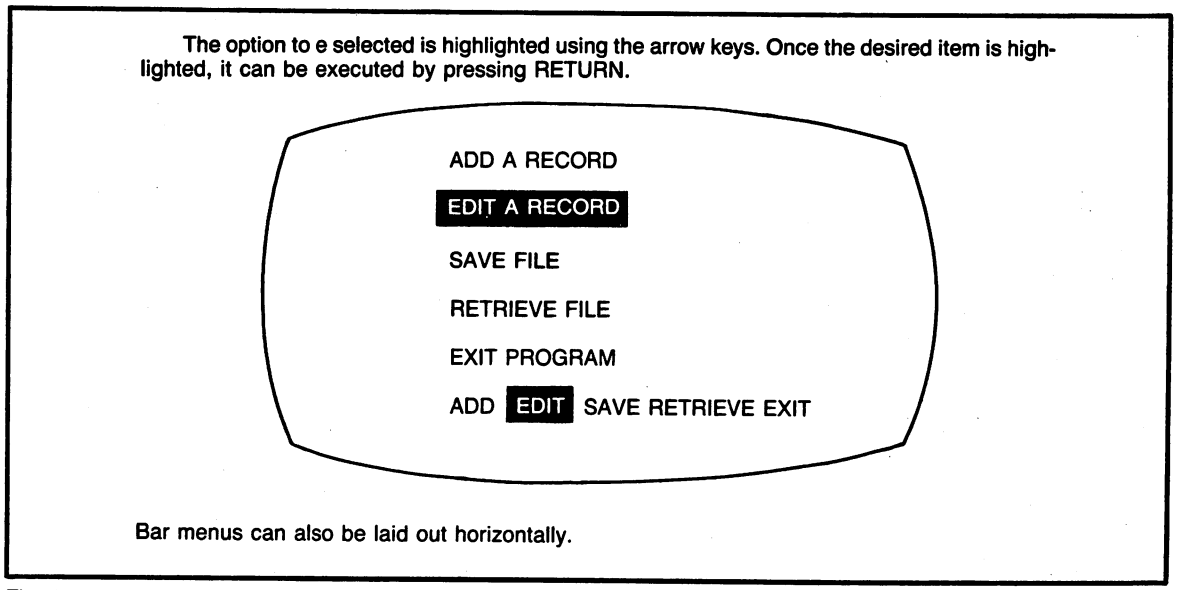


Fig. 9-9. A typical bar menu.

Ideally, the bar menu routine should advance the bar to the top of the menu if the bar is at the bottom when the down arrow is pressed. The bar should move to the bottom rung when it has been resting at the top and the up arrow is pressed.

Coding the Bar Menu

Let's begin with the first step, which is the routine to assign menu information. There are lots of ways this can be done. You could assign menu options directly: MN$(1)="PRINT INFORMATION". You could read these options from a file, or you could read them from DATA statements.

You'll notice that so far in this book we have not used data statements. This is because data statements can become troublesome. If there are several different types of data to be read, the program has to "skip" unneeded information until it comes to the right place in the data statement list. All this skipping calls for extra code and creates general confusion. For multiple applications, data statements are not really workable.

Data statements, however, can be immensely useful if you make a starting decision—and stick to it—that they will be used for one application only. Data statements are especially helpful when you're designing menus, because they work at lightening speed and enable you to easily change the names of menu options. If you've done any serious programming you already know how often menu options can change.

The data statements we use for the menu will contain:

1. The names of menu options, exactly as they should appear on the screen.
2. An end-of-names marker at the end to signify that there are no additional entries.

The system will use the marker as an indication that it should stop reading. Here's what the data statement might look like:

```
10000 DATA "ADD INFORMATION",
      "CHANGE INFORMATION",
      "SORT INFORMATION","###"
```

Note that menu options are enclosed in quotes as a matter of convention; they don't actually have to be in this case, because they've got no commas or colons. Generally, though, programmers will enclose any alphanumeric data items within quotes.

The program will keep a counter of how many menu items exist. In this way, the system can know instantly how many items are in the menu.

Reading in New Information

Figure 9-10 shows the routine that reads the information is very simple; it uses a loop that places new menu options in the array until the end-of-menu marker is encountered. Menu items are stored in the array SELECTS(x). The routine also adds column and row positions for each item name, so that menu selections will be indented on the screen.

The first step, which is to read the information, is now complete.

The Screen Display

Now that the options have been loaded into the array, the menu can be easily displayed on the screen, using the code shown in Fig. 9-11. You'll remember from our previous discussion of the CHAR command that the display can be highlighted simply by specifying a ,1 after the string to be printed. This feature can be used to no small advantage in our menu routine, because it allows us to set a switch that will turn the highlighting on or off for individual menu items. The highlighting will depend on the current item the cursor is pointing to.

The first portion of the display routine (lines 52000 through 52050) is dedicated to the displaying of the information on the screen for the first time. The routine at 54000 is then used to highlight the currently selected item (usually the first item) so that some option will be illuminated from the instant the menu appears on the screen. It will keep things esthetically proper. Once the full menu is displayed we'll take one item at a time. The full menu will not be redisplayed with each keystroke for this would take far too long, and would make operation painfully slow.

Instead, we'll have a separate routine that actually remembers the previously selected item and knows the current item. This will allow the system to:

- "Unhighlight" or turn the switch off, for the menu item that was last selected.
- Turn it on for the newly selected item.

The routine at line 54000 works beautifully, because only the two lines involved are reprinted; the computer does not have to refresh the rest of the screen. Figure 9-12 shows how menu items are highlighted.

Determining What Key Was Pressed

The most important thing is to keep the routine always ready to receive keystrokes and to properly interpret them. The arrows should be intercepted (to move the bar). Other keys (such as RETURN) should also be interpreted.

As we've pointed out before, each C-128 key has a specific ASCII code that can be detected. The first step is to store keystrokes in a form that can be easily tested. You'll remember that in Chapter 7, the GETKEY command was used to accept single keystrokes. The same trick has applications in the menu routine, which includes the GETKEY A$ statement at line 53010.

This routine will:

1. GETKEY the key from the keyboard.
2. Test to make sure it's one of the proper keys.

```
50000 : rem menu using on...gosub
50001 : read index$ :rem index to keys
50002 : do while select$(ctr)<>"###"
50003 :    ctr=ctr+1
50005 :    read col(ctr),row(ctr),select$(ctr)
50006 : loop
```

Fig. 9-10. A routine to read menu items from DATA statements.

```
0 :        goto 10
1 :
2 :  menu using on...gosub
3 :
4 :
10 ::::::::::::::::::::::::::::::::
20 : rem definitions
25 :
26 scnclr
30 dwn$=chr$(17)
40 up$=chr$(145)
50 retrn$=chr$(13)
60 esc$=chr$(27)
80 :
90 goto 50000
95 ::::::::::::::::::::::::::::::::
100 print:print"1 selected":sleep 1:return
200 print:print"2 selected":sleep 1:return
300 print:print"3 selected":sleep 1:return
400 print:print"4 selected":sleep 1:return
500 :
600 :
50000 : rem menu using on...gosub
50001 : read index$ :rem index to keys
50002 : do while select$(ctr)<>"###"
50003 :    ctr=ctr+1
50005 :    read col(ctr),row(ctr),select$(ctr)
50006 : loop
50007 : no=ctr-1 :rem nbr of items
50009 : do until esc
50010 :  gosub 52000   :rem display full
50012 :  choice=1:last=no
50015 :  gosub 54000   :rem display highlgt
50017 :  retrn=0:esc=0
50020 :  do until retrn or esc
50030 :     gosub 53000 :rem get key
50040 :     gosub 54000 :rem display partial
50050 :  loop
50090 :
50095 char ,0,22
50100 :   on choice gosub 100,200,300,400
50132 :  if choice=5 then esc=1
50133 ::::::::::::::::::::::::::::::::::::
50135 loop
50140 print:print"bye!       "
50150 end
```

Fig. 9-11. A complete bar menu program.

```
52000 : rem display all options
52010 :
52020 : for ctr=1 to no
52030 char ,col(ctr),row(ctr),select$(ctr),0
52040 next
52050 return
52060 :
53000 : rem   keypress
53005 : last=choice
53010 : getkey a$
53012 :
53015 if instr(index$,a$) then choice=instr(index$,a$)
53020 : if a$=up$ then choice=choice-1
53030 : if a$=dwn$ then choice=choice+1
53035 if a$=esc$ then esc=1:choice=no
53040 : if a$=retrn$ then retrn=1
53045 : if choice<1 then choice=no
53047 : if choice>no then choice=1
53048 :
53050 : return
53060 :
54000 :rem display bars
54010 :
54020 char ,col(last),row(last),select$(last),0
54030 char ,col(choice),row(choice),select$(choice),1
54035 char ,9,22,"esc to end"
54040 return
54050 :
60000 :rem  menu data
60015 data "adsie" :rem first ltr index
60020 data 5,10,"   add a record      "
60030 data 5,11,"   delete a record   "
60040 data 5,12,"   search this file  "
60050 data 5,13,"   index items       "
60060 data 5,14,"   end program       "
60200 data 0,0,"###"   : rem  end mark
```

3. Set the bar-up or bar-down flag.
4. Or set other flags.
5. RETURN from the subroutine in order to highlight the selected item.

The program will continue to access the keypress and display-bars routines from the loop in lines 50020 through 50050 until the user presses RETURN, at which time a special flag will be set and the system will "freeze" all selected variables exactly where they are.

Going Further

Once you have your bar menu working, you can add bells and whistles. For example, you might

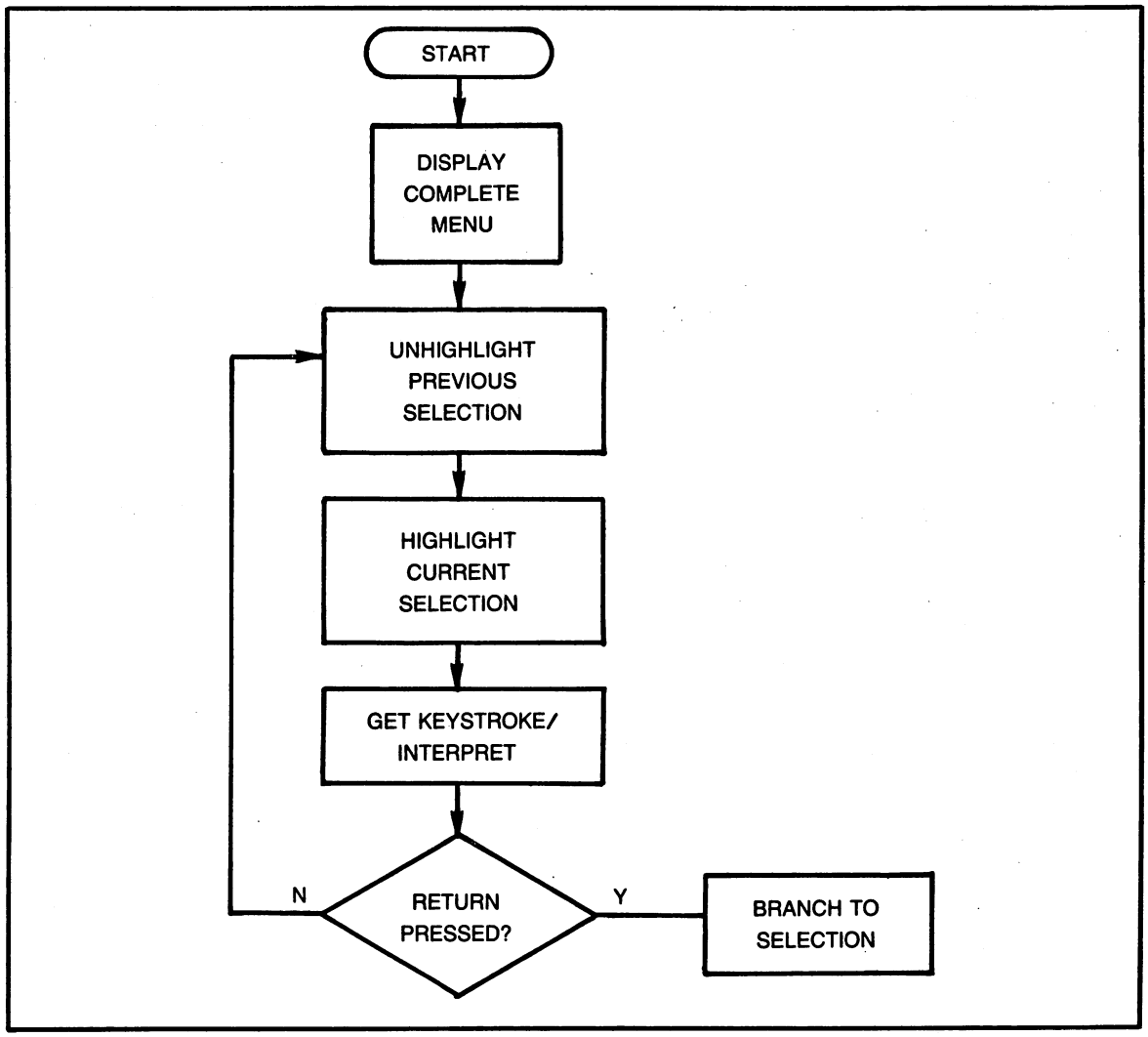


Fig. 9-12. How a bar menu works. All bar menus follow these general principles.

1. Add the option in the menu Data statements.
2. Change the ON . . . GOSUB line to include the New Menu Option.
3. Test that the new option runs from the MENU, and that it returns to the menu once it has executed.

It is generally recommended that your menu titles be action oriented, using verbs. For example, "ADD A RECORD" is a lot better than "RECORD ADDITION."

Fig. 9-13. Steps for revising a menu.

want to intercept the help key (or the first letter of it, H) to provide explanations of menu options.

As done in Listing 9-11, the ESC key might be intercepted, allowing users to quickly exit the program.

Finally, you might want to include a routine for some fancy headings, so that all your menus can have a standard polish to them. Figure 9-13 shows the steps involved in revising a menu.

Chapter 10

Professional Programming: Speed and Readability

There are a lot of old wives' tales about BASIC programming . . . and they die hard. One truism is that for optimum speed, you must place your least-used routines at the bottom of the program and your most-used applications at the top.

Another says that too many remarks within a program will slow its operation to a crawl. Yet another states that the less readable (more compact) a program is, the faster it will execute.

On the Commodore 128, all of these assumptions are false. In fact, following some of them will slow down your program instead of speeding it up! These old wives' tales about programming stem from the distant past, when computers and BASIC were not nearly as sophisticated.

WHAT WORKS, WHAT DOESN'T

To be sure, there are guidelines that will speed operation and make your programs more manageable. We'll talk about them extensively in this chapter.

As you'll see, though, the old wives' tales will waste precious time. It does not necessarily pay to rearrange your program so the most-used routines are at the top. It does not necessarily speed up execution to remove all REMs from your program. And the slight efficiency to be gained by compacting a line beyond all readability is cancelled out by increased enhancement and debugging time.

There are some tricks to learn, however, about speeding up your programs and taking full advantage of the C-128's flexibility. The Commodore has a set of rules all its own—rules you won't find in the manual.

Perhaps the best way to see what slows down a program, and what doesn't, is to review how BASIC operates: to see how it finds lines, how it keeps track of where it is, and how it knows where it's going.

When a program is loaded or typed in from the keyboard, the computer automatically makes a mental note of the program's beginning memory location. BASIC stores these points in an out-of-the-way area so it can quickly go to the top of the pro-

gram at any time. (In case you're curious, the memory location that points to the beginning address of the program is 4528 decimal, or $2D hex.)

Usually, BASIC has no reason to be concerned about where the program starts. PRINT, INPUT, and most other commands operate in the same way no matter where they are in the program. But there are certain statements that force the computer to go to the top of the program most of the time. When the program seeks a line such as this:

```
200 GOTO 100
```

you probably think the computer instantly knows where line 100 is and jumps to it immediately. Nothing could be further from the truth. Actually the C-128 jumps to the top of the program and commences a long process of counting ahead down to line 100.

Every line between the beginning of the program and the final destination must be examined for a match. If the program doesn't find the line, of course, an UNDEF'D STATEMENT error occurs, and the program stops.

Because of the way lines are stored internally (in the most compact form available to the computer), there's no *index* to tell the machine where to find a line number. So it dutifully examines and skips line numbers until it arrives at its destination. Then the statement is executed.

BASIC is itself written in straight machine language, of course, so this is a pretty quick procedure. But the time can add up. If BASIC is skipping a large number of lines (say 1,000), you can actually count a beat or two before the program reaches the desired line and reacts.

Here's another way to picture the situation. Imagine for a moment an unusual Halloween game. When children stop at a house for candy, they are given the address of the next house they must visit. If the new address is up the road (greater than) their current address, they can proceed. But if it is down the road, over pavement they've passed already, they'll have to go to the beginning of the road before they can head for the new house. It's a silly game, but it's exactly how BASIC moves through programs.

BASIC's way of doing these things adds up to a complication we don't usually plan for in designing our programs: routines placed at the bottom of the program that use GOTOs extensively can be much slower than if they are placed at the top of the program. The solution, however, is not to rearrange your entire program. There are lots of alternatives to GOTOs.

You already know about three of them: DO/WHILE, DO/UNTIL, and FOR . . . NEXT.

Let's look at what happens in a typical DO/UNTIL loop. Figures 10-1 and 10-2 illustrated that this operation,which counts from 1 to 1000, could be done effectively with either GOTO or DO/UNTIL. The GOTO style of loop, however, takes appreciably longer if there is a good number of program lines above it.

Here's why: When a GOTO that directs the computer to a previous line number is encountered, the machine must go to the top of the program and count down to the number specified. In Fig. 10-1 the computer has no way of knowing that the line it wants (the line to GOTO) is only three or four lines above, BASIC must go to the beginning of the program and count down to locate the proper line, as illustrated in Fig. 10-3.

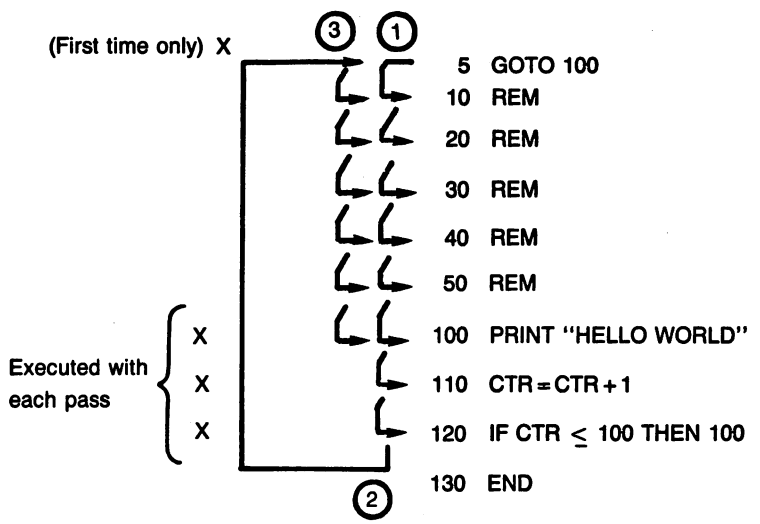
Now, as shown in Fig. 10-4, here's what happens in a DO/UNTIL loop: when a DO/UNTIL, DO/WHILE or FOR . . . NEXT loop is executed, BASIC makes a mental note of the loop's starting

```
10 : rem  loop with goto
20 :
1000 ctr=ctr+1
1010 :
1020 :
1030 print "pass # ";ctr
1040 :
1050 if ctr<=900 then 1000
1055 :
1060 end
1070 :
```

Fig. 10-1. A loop using GOTO.

```
10 : rem  loop with do/until
20 :
1000 do until ctr=900
1005 :
1010 ctr=ctr+1
1020 :
1030 print "pass # ";ctr
1040 :
1050 loop
1055 :
1060 end
1070 :
```

Fig. 10-2. A loop using DO/UNTIL.



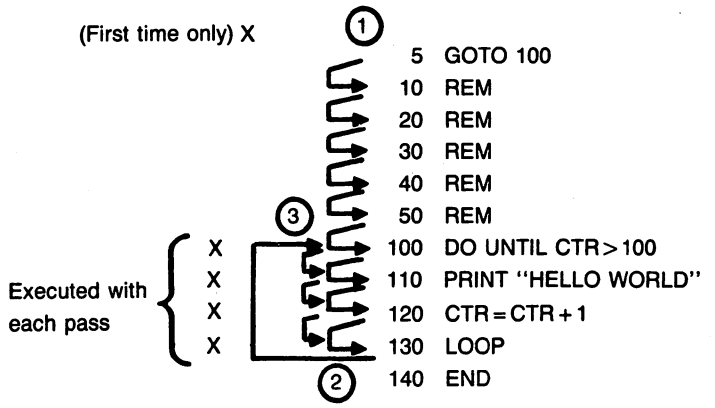
1. The program starts by counting up the lines until it reaches 100.

2. Once lines 100 and 110 are executed, the computer is instructed to branch back to 100.

3. Even though line 100 is just two lines away, the computer must go to the top and begin "counting up" again.

Note: Only lines marked with an X are executed. All others are skipped over.

Fig. 10-3. Diagram of loop operation using GOTO.



1. The program starts by counting up the lines until it reaches 100.
2. Once lines 100, 110 and 120 are executed, the computer is instructed to loop back to the beginning.
3. Because the DO/UNTIL and DO/WHIIE structures remember where the loop starts, the computer goes directly to line 100 and continues. FOR . . . NEXT works under the same principles.

Note: Only lines marked with an X are executed. All others are skipped over.

Fig. 10-4. Diagram of loop operation using DO/UNTIL.

line. When it encounters the LOOP or NEXT command, it simply jumps back to the memory location it noted at the beginning of the loop. Since this loop-start address is a direct memory location, there's no counting involved. The computer's mental note tells it immediately which line to go back to.

It's as if our Halloween kids could go back to a house immediately, as long as they'd planted a flag in the front yard.

You can see why DO/UNTIL, DO/WHILE, and FOR . . . NEXT loops can be much faster than GOTOs, even if they're essentially performing the same operations. Not only are the loop statements easier to read; they are also easy to execute—a case of having your cake and eating it too.

If you're really a speed demon, here's another item for your programming design list: DO and FOR . . . NEXT loops also work at slightly different speeds. Which one do you think is faster?

FOR . . . NEXT Structures

Throughout this book we've used DO/LOOP structures extensively. Partly that's because they're attractive and easy to read. Partly too it's because it is fun to play with new types of statements. Give programmers a new set of commands, and the new commands will be applied in every situation, regardless of how advisable that may be.

But let's not forget about our old friend FOR . . . NEXT. It's been in BASIC from the beginning, and as you've seen in the previous example, its operation on the Commodore 128 is quite fast.

In fact, for a loop that involves simple counting, FOR . . . NEXT beats the DO/LOOP structure hands down. Figures 10-1 and 10-2 show two such loops, and you can readily see why rewriting the loop using FOR . . . NEXT would have the edge. There's an extra statement required for the DO/LOOP structure: CTR=CTR+1. When mul-

tiplied thousands of times, that simple statement is like running on regular versus hi-test—the DO/LOOP loses every time. It's not much of a difference, but if you're using a look for thousands of passes you should certainly keep it in mind. If you take the trouble to time-test these routines, you'll find that the FOR . . . NEXT structure is about ten percent faster.

MAKING THE MOST OF SUBROUTINES

What about subroutines? Would a program that makes extensive use of them be slowed down, or not?

The facts might surprise you. If things are handled properly, programs with lots of subroutines tend to be a lot faster than programs without a lot of subroutines.

Rule 1. Always call your subroutines from above:

```
100  GOSUB 150
110  GOSUB 120
```

The Commodore 128 always recognizes the current line number, so it can quickly search forward for a line without starting a long trip from the top of the program. If a destination is 10 lines ahead, BASIC will only have to examine 10 lines to find a match. This means that setting to a line with GOSUB works in much the same way it does with GOTO.

```
10 :
20 :
30 :
40 :
900 gosub 1000
950 :
960 end
970 :
980 :
990 :
1000 print"hello"
1010 return
```

Fig. 10-5. Calling a subroutine that's below.

Rule 2. Group frequently used subroutines directly under the routine that calls them.

Let's take a look at how this works.

Figures 10-5 and 10-6 show two methods of calling a subroutine. One places the subroutine directly below the routine that is calling it. The other places the routine somewhere in the middle of the program. Now, let's go through what happens when each of these programs is executed.

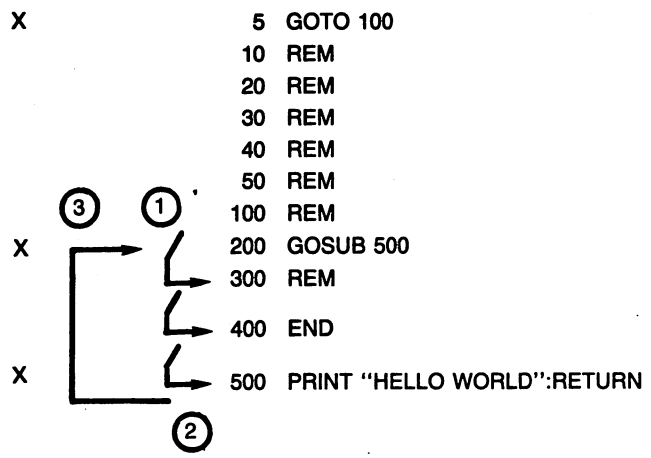
In the first example, when the computer hits the GOSUB it immediately determines that the line number is greater than the current position—the line number is further down in the program. So it skips down ten or twenty lines to that routine very quickly.

In the second example, the computer determines that the line number is above it. Since BASIC can't count backwards, it does the only thing it knows how to do. It goes to the top of the program and begins counting down, perhaps skipping over as many as one hundred line numbers before lighting on the desired routine. This approach is very slow.

As shown in Figs. 10-7 and 10-8, in both cases, the program goes back to the original routine immediately (without counting lines), because the

```
5 goto 2000 :rem main processing
10 :
20 :
30 :
40 :
1000 print"hello"
1010 return
2000 :
2010 :
2020 :
2030 :
2040 gosub 1000
2050 :
2060 end
2070 :
2080 :
2090 :
```

Fig. 10-6. Calling a subroutine that's above.



1. Once the computer arrives at line 200, it interprets an instruction to perform the routine at line 500.

2. Because line 500 is greater than 200, the C-128 simply counts ahead three lines.

3. The computer executes the routine at line 500, and returns directly to line 200 (it remembers where the subroutine call originated).

Note: Only lines marked with an X are executed. All others are skipped over.

Fig. 10-7. Calling a routine that's at a higher line number.

Commodore 128 keeps a *stack* of originating positions for GOSUBs. Like a homing pigeon set free far from base, the program always knows exactly where to go when it encounters a RETURN statement.

Both approaches work, but the first is much faster, especially when the routine is being called several times. The key is placing the routine being called as close as possible to the main routine. If the routines being called were far below, getting to them could take just as long. As always in computing, the split seconds do add up.

In review, DO/LOOP and FOR . . . NEXT statements are the fastest way to go, and GOSUBs should call lines deeper into the program (lines with higher numbers), whenever possible. These two tricks will save lots and lots of time.

PROGRAMS THAT ARE REMARKABLE

One of the other common falacies of speed programming is that the addition of REMarks will appreciably slow down the program. Actually, if you use the tricks outlined so far, REMs should cause you no trouble at all, and they're essential for program readability.

If you're truly concerned with extracting the last ounce of speed from your machine, removing some types of REMarks can help a little. REMs that don't require a separate line number obviously mean one less line number for BASIC to skip over when it's hunting down a routine. There's a way to remove the REMs without sacrificing readability.

While REMs on separate lines can slow down a program, REMs placed at the end of existing information do not. It's also important to realize that

BASIC skips all information following a remark. So if you place a REM at the end of a line that would have existed anyway, you're not slowing things down. After executing everything before the REM, the Commodore 128 skips to the next line:

```
20 PRINT "HELLO THERE ";A$: REM
   A$ IS NAME
30 PRINT: REM CONTINUE WITH
   PROGRAM
```

Therefore, if you are very, very concerned about speed, simply place all your REMs at the end of existing statements on existing lines.

OTHER SPEED HINTS

Here is where we really sacrifice. You may really want to reserve the following program turbochargers for the routines you use most often—where speed really *is* crucial. Most of these tricks sacrifice program readability for faster operation.

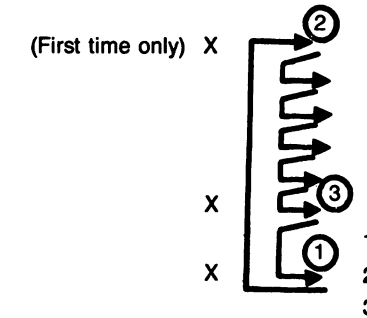
One of BASIC 7.0's stronger features is that you can use long, descriptive variable names. You've seen it a lot in this book. Only the first two characters count, so SAVINGS and SAL would refer to the same variable. Still, it's nice to use variable names that have some character.

Unfortunately, long variable names slow down your computer like a load of excess baggage. Because BASIC interprets each character individually, long variable names take longer to process than short variable names. Again, it becomes a compromise between speed and readability:

```
10 SAVINGS=INCOME-COSTS
```

could be written as:

```
10 SA=IN-CO
```




```
5   GOTO 100
10  REM
20  REM
30  REM
40  REM
50  PRINT "HELLO WORLD":RETURN
100 REM
200 GOSUB 50
300 REM
400 REM
500 END
```

(1) Once the computer arrives at line 200, it interprets an instruction to perform the routine at line 50.

(2) Because line 50 is lower than 200, the C-128 must return to the top of the program and begin counting upward.

(3) The computer executes the routine at line 50, and returns directly to line 200 (it remembers where the subroutine call originated).

Note: Only lines marked with an X are executed. All others are skipped over.

Fig. 10-8. Calling a routine that's at a lower line number.

Even though the performance difference is not great, if you don't mind sacrificing readability, you can shorten variable names in some of the more speed-critical areas of your program.

Remember that both the first and second letters of variable names are significant. SA is not the same as S, but SA and SAVINGS will contain the same values.

Here's another sacrifice you can make. Throughout this book you've seen the trick of indenting program lines by starting them with a colon:

```
10 :     REM  :THIS IS A TEST:
20 FOR X=1 TO 10
30 :
40 : PRINT "THIS IS NUMBER"
50 : PRINT X
60 :
70 NEXT
```

This indentation makes loops and a few other program structures much easier to read. It also enables you to set REMs apart from the main body of the program. But indentation adds spaces, and spaces take time to decode, even though they contain no useful information. Removing spaces from programs make them operate at a faster clip (and saves storage space to boot).

If you want to remove the extra space caused by indentation in such lines, simply delete the colon. BASIC will eliminate leading spaces on program lines automatically, By the way, the extra lines created by colons that fly solo (lines 30 and 60) also delay the program for the same reason as solitary REMs do. If you really want to travel light, these should be eliminated as well.

Speed Ups with Variables

One of the older tricks in giving BASIC a little extra kick is to replace commonly used words and numbers with variables. When BASIC encounters a print or processing statement such as:

```
A = 7 * 3
```

It must interpret and translate the 7 and 3 into its own binary number system. Both operations take time. However, when the computer encounters the same values in variables:

```
A = B * C
```

B and C having been assigned as 7 and 3 respectively, processing time diminishes, because BASIC automatically knows the values of the variables, which are kept in a special variable table in the memory. The Commodore 128 always knows where those variables are located and their value. What's more, the value has already been translated into the computer's native binary number system. This fact makes A=B*C faster than the perhaps more readable alternative, A=7*3.

The same holds true for strings:

```
IF Y$=A$ THEN 1000
```

is faster than:

```
IF Y$="YES" THEN 1000
```

These variables should be assigned near the top of the program, because the computer accesses variables in the order in which they were defined. Placing definitions at the top (or in a subroutine that's called from the top) ensures that BASIC will find the variable more quickly.

Garbage Collection

Many programs written for other computers

```
10 rem :compacted example of perfect hostess program:
20 print"would either of you like more coffee?":
   input"husband's response";h$:input"wife's response";w$
90 ifh$="yes"orw$="yes"thenprint"i'll make some then"
```

Fig. 10-9. An example of a compacted program.

shy away from routines, such as our input function described in Chapter 7, which continually store information into the same variable over and over again. (In that input routine, each character typed is stored in CH$—with CH$ being assigned and reassigned literally hundreds of times.

There's good reason for this precaution on many computers, because a process called garbage *collection* does an electronic inventory of memory, clearing out variables which have been reassigned several times. Garbage collection kicks into action without warning, and on many computers—even expensive business systems—this process is painfully slow.

The Commodore 128 features a very fast garbage collection routine, which does its work in the blink of an eye, even if variables have been reduplicated hundreds of times. Therefore, your program routines can use the same variables over and over, and you'll seldom notice the computer's refuse collection operation.

If you wish to avoid garbage collection in a speed-critical routine, simply add the following statement at a point in the program somewhere before the routine is called:

```
X=FRE(1)
```

This statement can be used to force garbage collection at any point in a program.

HOW TO TELL IF A CHANGE WILL HELP

There's no reason to rearrange your program for speed unless you're certain a particular change will help. The changes outlined in this book will generally increase a program's speed measurably, but you may come up with ideas of your own that could also make a difference in execution time.

You can perform head-to-head comparisons on different versions of almost any type of routine for speed by placing them in a loop of a few thousand cycles. If you're interested in how extra line numbers will affect the operation, place about 200 extra solo colon lines at the top of the program (this is most quickly done using the AUTO command):

```
  5 FOR X=1 TO 3000:GOSUB 300:NEXT
 10 :
 20 :
 30 :
 40 :
 50 : REM ETC
300 REM BEGIN ROUTINE
```

Line 5 contains a loop that calls the routine to be tested 3,000 times. To compare speed, change the routine at line 300, trying the different versions for speed. Test out two types of similar routines in this manner, and you'll soon know which is faster.

FAST VERSUS SLOW

There's a final gem which the 128 has given us to make things go really FAST. That's the name of the command: FAST. It works by switching into a special high-speed two-megahertz mode, and can almost double the speed at which some operations are performed. But it comes at a price. When FAST is activated, the 40-column screen will go blank. You won't see anything at all—not even what was just on the screen.

Therefore, FAST is not really appropriate for speeding operations if you also want to tell the user what is going on. Messages like "NOW SORTING" simply won't be visible. The display returns to normal when the SLOW command is used. Anything printed to the screen while FAST was active will be shown.

Also, FAST does not speedup I/O operations. That means that disk access, printing, keyboard entry, and so on work just as well in the SLOW (normal) mode of the machine.

Chapter 11

Professional Design: Error Trapping

Crash is the most dreaded word in program operation. For users it means that a program has halted abruptly, and there's no easy way to get back in. Crashes can come about in hundreds of different ways. From improper keyboard entry to read errors on bad disks, they're a part of life. Fortunately, the C-128 is well-equipped to intercept errors before they become a problem for your programs and their users.

We've already seen how certain errors can be trapped before they're able to do any damage. For example, in the section on input, you were shown how to screen each character as it is entered from the keyboard, eliminating characters that could cause problems in file routines or that should not be accepted by the program.

This is the most important form of error trapping. If your programs are galvanized with routines that reject improper entry, and if they anticipate the user's questions and mistakes, you're 90 percent of the way toward eliminating all crashes.

There are, however, certain types of errors that keyboard screening and instructions to the user simply cannot take care of. What happens if the operator presses the RUN/STOP key (which halts the program) in the middle of a crucial section of the program? What happens if the user opens the drive door? Or removes the disk when the program is trying to read information from a file?

How do you handle ?SYNTAX ERRORs in the program? Should operation come to a halt? Or should you indicate the line number of the error, display it, and then return to the main menu?

The answer is that you want to screen all of these possible errors and still allow the user the greatest freedom in operation your software. The TRAP command provides the means by which you can accomplish these goals.

THE TRAP COMMAND

The start of the show is the Commodore 128's TRAP command, which instructs the computer to trap any error encountered. When TRAP is active, and the computer encounters an error, the operation jumps to a specific routine designed expressly

to handle errors.

```
10 TRAP 3000 :REM GOTO 3000
   ANYTIME THERE'S AN ERROR
20 PRINT "HELLO"
30 :
40 :     REM REST OF PROGRAM
50 :
3000 :   REM ERROR TRAPPING
   ROUTINE
3050 PRINT "YOU HAVE AN ERROR"
3060 GETKEY A$
3070 RESUME
```

The TRAP command works under a principle that programmers refer to as *event trapping*. It works like this: instead of testing for something at each line, you can simply say, "If there is an error anywhere in the program, go to the routine at line 3000 and perform the routines there."

Other BASIC commands are used in conjunction with TRAP. There are commands to determine what error was encountered and to continue operation where the program left off, and there's even a command to resume operation at a completely different place. If you do it right, you can handle any error without missing a beat.

Because the Commodore 128 always "knows" what error has occurred (through special variables that contain error codes), it is possible to provide very specific messages about what error was trapped and to display further instructions. Your programs can be made to seem intelligent, to say things like "Please reinsert your disk!"

In addition to the message you devise yourself, the Commodore 128 also has two special string variables, ERR$ and DS$, which display messages for the error encountered.

In the previous example, you saw how TRAP is used to intercept errors. Here's another example, using the ERR$(x) function (line 3060) to display an error description:

```
10 TRAP 3000 :REM GOTO 3000
   ANYTIME THERE'S AN ERROR
20 PRINT "HELLO"
30 :
40 :     REM REST OF PROGRAM
50 :
3000 :   REM ERROR TRAPPING
   ROUTINE
3050 PRINT "YOU HAVE AN ERROR"
3060 PRINT ERR$(ER)   :REM SHOW
   ERROR MSG
3065 GETKEY A$
3070 RESUME
```

Let's take a moment to examine this program. First, at line 10 we've instructed the computer to branch to the error routine anytime a problem is encountered. We have also included an error routine that not only traps the error, but prints what is wrong. After pausing for the user's input, this routine resumes with current program operation.

No matter what type of error trapping you're doing, you'll probably use RESUME at least once in your error routine. You can tell the computer to resume at a specific line number:

```
RESUME 1970
```

or even to resume operation at the statement following the one that caused the original error.

The error message displayed when you've divided a number by a zero value would look like this:

```
DIVISION BY ZERO
```

It's pretty straightforward, even if it does leave a little to the imagination. Disk error messages are somewhat more mysterious:

```
62,FILE NOT FOUND,00,00
```

The statement tells a story, as long as you're good at reading code; Disk error number 62 has occurred—the file was not found. To a beginning user, this message can be quite confusing. *Why* wasn't the file found? *What should the operator do?* Fortunately, there's a simple solution, because you can strip these codes away and elaborate further.

MAKING ERROR MESSAGES MORE READABLE

The heart of the statement listed above is the message itself. The numbers on either side (listing error codes, etc) would be of interest to few program users. They would confuse the users more than they would help them.

There's a standard set of statements you can place in your error routines to eliminate these special codes and display only the text portion of the error messages. Placed in our previously listed program, it would look like this:

```
3051 IF ER> 10 AND ER< > 41 THEN
     MSG$=ERR$(ER):ELSE BEGIN:    :REM TEST FOR NON-DISK ERROR
3052 MSG$=DS$                     :REM ASSIGN ERROR MSG
3054 X=INSTR(MSG$, "," ,4)        :REM FIND LAST NUMBER
3056 MSG$=MID$(MSG$,4,X-4)        :REM MESSAGE BETWEEN
                                   NUMBERS
3057 BEND
3060 PRINT MSG                    :REM SHOW ERROR MSG
3065 GETKEY A$
3070 RESUME
```

Lines 3052-3060 extract the text from the standard error message and then display this text with no numbers included. Line 3054 instructs the computer to scan the error message for a comma, starting at the forth position. Let's take a look at the standard system error message again:

62,"FILE NOT FOUND",00

Since the first two commas and numbers are to the left of the forth position, INSTR picks up only the position of the last comma in the message, which also happens to be the marker for the end of text. Line 3056 uses the X variable returned by this INSTR scan to determine where text ends.

This routine is only really needed for occasions when a drive error has occurred, since only drive error messages produce the funny numbers to the left and right of the message.

A STEP FURTHER: LETTING ERRORS MAKE DECISIONS

Let's take a look at a few facts that taken together make the Commodore 128's error trapping feature a supremely powerful way to glide over program errors and help the user avoid them:

Fact 1: The Commodore 128 returns a specific code for almost every error encountered.
Fact 2: The Commodore 128 allows you to test the variable containing this code, just like any other variable.
Fact 3: Your program can branch to different subroutines based on what code your error test uncovers.

Because all errors generate a code, and because similar error codes are grouped together, it is very easy to test for specific errors or ranges of errors. Figure 11-1 lists typical errors and the action a program should take when it encounters them. We'll cover each of these occurrences in the next few pages.

DISK ERRORS

There are many things that can cause problems with a disk operation. If the user removes a disk from the drive before or during file access, an error will occur. If the computer tries to open a file for writing when a write protect tab is covering the disk notch, an error will occur. If your program runs another program, and that second program is not on the disk, still another error will occur. The Commodore 128 uses specific codes to determine which disk error has occurred. You can use these codes when having the program make decisions about what to do to remedy these errors (Fig. 11-2 shows a list of disk error codes). There's even a general

Number	Message	Action
1	TOO MANY FILES	Close one of the files in your program.
2	FILE OPEN	Close files after using them; use a different file #.
5	DEVICE NOT PRESENT	Turn on printer, disk drive, etc. be certain it is properly connected.
6	NOT AN INPUT FILE	Change the OPEN statement to allow a read.
7	NOT AN OUTPUT FILE	Change the OPEN statement to allow a write.
10	NEXT WITHOUT FOR	Trace through logic of your program; look for GOTO branches.
12	RETURN WITHOUT GOSUB	Be certain your program finishes with an END if there are subroutines below the main section of your program.
13	OUT OF DATA	Count up the DATA items—they should match the number of READs. Be sure all lines of data are preceded by a DATA statement. Always issue a RESTORE before rereading the same data in a program.
16	OUT OF MEMORY	Determine available memory using FRE(0) and FRE(1). IF FRE(0) is low, issue a GRAPHIC CLR command. If FRE(1) is low, reduce the size of dimensioned variables. Also check for a large number of nested DO, FOR, or GOSUB statements.
18	BAD SUBSCRIPT	Dimension this array to a larger size. Remember that double dimensioned arrays cannot be referred to as single dimensioned, and vice versa.
19	REDIM'D ARRAY	Do not dimension an array more than once.
20	DIVISION BY ZERO	If the divisor is zero, skip the formula in question: IF A=0 THEN C=0: ELSE C = B/A.
22	TYPE MISMATCH	Check the statement carefully. Be sure to use STR$ when concatenating numbers to string variables.
35	NO GRAPHICS AREA	Issue a GRAPHIC command before using DRAW, CIRCLE, or BOX. You must use GRAPHIC 1, 2, 3 or 4 for this command.

Fig. 11-1. Types of errors and the most desirable responses.

way of testing whether the error encountered was a disk error or some other type of mishap.

Because the TRAP command does not intercept most disk errors, your programs should test for disk errors after each routine that accesses the disk (read, write, directory, and so on). Remember that most disk errors won't stop your program like other program problems will (disk errors simply cause the drive light to flash). Therefore, you can test for a disk error at any time after a disk-related operation has been performed.

There's no end to the treatments that could be applied to disk errors. You could display instruction messages on the screen, attempt to resave information, or even format a disk—all depending on the type of error that has occurred.

Perhaps the simplest approach to disk errors is to simply display a statement instructing the user to check the drive:

```
THERE IS A DISK ERROR!
PLEASE CHECK YOUR DRIVE
PRESS RETURN TO CONTINUE
```

Depending on the program's design and purpose,

Number	Message	Action
20-24, 27	READ ERROR	Retry or return to menu, switch to backup disks.
25	WRITE ERROR	Retry write; rewrite file on a different disk.
26	WRITE PROTECT ON	Display message to remove write protect tab.
29	DISK ID MISMATCH	Retry or initialize (Header) disk.
51	OVERFLOW IN RECORD	Screen the information being entered (to avoid strings that are too long); or expand size of records and recreate file.
60	WRITE FILE OPEN	DCLOSE the file and open it for reading.
61	FILE NOT OPEN	Check that the file has not been closed before this operation; be sure the file was opened to begin with.
62	FILE NOT FOUND	Respecify the file name; switch disks.
70	NO CHANNEL	Close a file temporarily.
72	DISK FULL	Close all files. Then SCRATCH obsolete programs and data files, or HEADER a new disk.
74	DRIVE NOT READY	Be sure a disk has been inserted and the drive lever is closed.

Fig. 11-2. Disk errors codes and the most desirable responses.

it can either rerun from the beginning or resume at a central program routine (such as a main menu).

How to Tell When an Error Occurs

The disk error variable DS can tell you whether or not a disk error has occurred, and if so, what type of error it was. When DS is greater than or equal to 20, you can be assured that some type of disk error has occurred.

Several disk errors are also reflected in the ER variable as well, so it's possible to have a file-not-found error that can be intercepted by both DS and ER. (Include DS in your programs only when you know there will be a disk drive connected, by the way, or you'll get an error trying to use this variable. It's directly dependent on the disk drive.)

Of course, the more specific your testing for errors, the more efficient and helpful the program can be. For example, if the write-protect notch is covered, you might want to display a separate message:

```
PLEASE REMOVE THE WRITE-PROTECT
LABEL FROM YOUR DISK.
THEN REINSERT THE DISK
AND PRESS RETURN.
```

If the disk were full, you could provide special instructions for formatting a new disk. You might even want to include a special routine that would header the disk—of course first checking that the disk did not contain information. This could be done by using yet another error code, which would try to open a temporary file to see if the disk had been formatted (if it hasn't been, you'll get a drive-not-ready error and the program can proceed with formatting.)

The best solution for error handling is a set of subroutines which groups errors according to their type and severity. Figure 11-3 shows a complete error-trapping routine.

TESTING FOR RUN/STOP

There's nothing worse than having a program stop because you've accidentally hit the wrong key. And RUN/STOP is easy to hit, nestled as it is next to CONTROL, SHIFT LOCK and SHIFT.

Normally, anytime the RUN/STOP key is

```
10 trap 59000
20 :
30 :
35 : rem simulate errors
36 catalog
37 if ds>19 then goto 59500
38 stop
40 syntax
50 next
60 a=a/0
70 return
80 a=1000000000^100000000
90 dim a$(1000,1000,1000,1000)
100 end
200 :
300 :
59000 ::::::::::::::::::::::::::::
59010 : rem  error routine        :
59020 ::::::::::::::::::::::::::::
59030 :
59040 : rem does not handle disk errors
59045 :
59050 scnclr
59060 label$="error!":gosub 63000
59065 if instr("aeiou",left$(err$(er),1)) then aa$="an ": else aa$=
      "a "
59070 char ,0,5,aa$+err$(er)+" error has occured."
59075 char ,0,7,"please press one of the following keys:"
59080 char ,6,10," r ",1
59090 char ,6,12," s ",1
59100 char ,6,14," m ",1
59110 char ,11,10,"- retry this operation"
59120 char ,11,12,"- skip to next operation"
59130 char ,11,14,"- return to menu"
59140 getkey a$
59150 if instr("Ss",a$) then resume next
59160 if instr("Mm",a$) then resume 50000 : rem menu
59170 resume :rem default
59499 :
59500 ::::::::::::::::::::::::::::
59510 : rem disk error traps      :
59520 ::::::::::::::::::::::::::::
59530 :
59550 scnclr
59560 label$="disk error":gosub 63000
59570 :
```

Fig. 11-3. A full trap routine.

```
59580 :if instr("20 22 23 24 25 27 28 29",str$(ds)) then begin
59590 :   char ,0,10,"there is a problem reading this"
59600 :   char ,0,12,"diskette. make sure you have current"
59610 :   char ,0,14,"backups."
59620 :   getkey a$:dclear: resume 50000
59630 :bend
59640 :
59650 :if instr("30 31 32 33 34 39",str$(ds)) then begin
59660 :   char ,0,10,"there is a dos syntax error"
59670 :   char ,0,12,"please check the file name and retry"
59680 :   getkey a$:dclear: resume 50000
59690 :bend
59691 :if ds=26 then begin
59693 :   char ,0,10,"this disk is write protected"
59695 :   char ,0,12,"please switch disks or remove the"
59697 :   char ,0,14,"write protect label"
59698 :   getkey a$:dclear: resume
59699 :bend
59700 :
59710 :if instr("72 52",str$(ds)) then begin
59720 :   char ,0,10,"the disk is full."
59730 :   char ,0,12,"Please insert another and try again"
59740 :   getkey a$: dclear:resume 50000
59750 :bend
59760 :
59770 :rem all other errors
59780 msg$=ds$
59790 x=instr(msg$,",",4)
59795 msg$=mid$(msg$,4,x-4)
59800 char ,0,10,msg$+" error!"
59810 :   getkey a$: dclear: resume 50000
63000 ::::::::::::::::::::::::::::::
63010 : rem print nifty screen head:
63020 ::::::::::::::::::::::::::::::
63030 :
63040 ll=len(label$)
63050 smark=20-((ll*4)/2)
63060 for ctr=1 to ll
63070 char ,smark,1," "+mid$(label$,ctr,1)+" ",1
63075 smark=smark+4
63080 next
63090 return
```

pressed, a BASIC program will come grinding to a halt. While this feature is great for debugging, it's not something you want people to do when they're operating your program. Data could be destroyed, the program could be accidentally altered, and, most of all, many users don't know how to

restart a program (with RUN or CONT) once they've stopped it using RUN/STOP.

Pressing RUN/STOP actually causes an error, which in turn creates an error code—so far, so good . . . But to trap the RUN/STOP key successfully requires special attention to program structure, because the C-128 can easily become confused on successive RUN/STOP attempts. First, the TRAP statement and TRAP routine should be the very first items in your program:

```
10 TRAP 12
11 GOTO 20 :REM START OF PRO-
   GRAM
12 TRAP 12  :REM ERROR TRAPPING
14 IF ER < > 30 THEN GOSUB 45000 :
   REM DO TESTS AND DISPLAYS
16 RESUME:RESUME
18 :
20 :       :REM START OF PROGRAM
```

You're right if you think this looks a little convoluted and unstructured. But it works. And it's the only way to be assured of trapping the RUN/STOP key successfully. To understand better what's going on in this spaghetti-plate approach to programming, let's take a look at each step.

Line 10 is easy; it tells the computer to go to line 12 anytime an error is encountered. The next statement, line 11, branches around the error routine (from lines 12-18) and goes directly to the main body of the program.

Why, you ask, should the error routine be thrown right in the middle of everything? Why not place it at the bottom of the program? And why is there a TRAP 12 statement at line 12, the very line that begins the error routine? All are good questions, to be sure. The answer lies locked in the way BASIC traps errors, and in the way it fails to trap the RUN/STOP key under certain circumstances.

The first thing to realize is that BASIC doesn't automatically know where every program line is located. Each time an error is encountered, and the computer must redirect itself to an error-handling routine, BASIC goes to the very top of the program and begins scanning downward for the appropriate line. So if there are one-thousand lines between the top of the program and the error trapping routine, BASIC will check each of those lines before hitting the error routine and executing it.

In the brief nanoseconds BASIC is tracking down the error routine, error trapping itself is turned off. The shields are down. The force field is out of order.

So what happens if the RUN/STOP key is pressed while BASIC is hunting down the error routine's line number? Quite simply, the program crashes to a halt. This kind of crash is quite possible in a large program in which the error trapping routine has been placed at the bottom, because there is a significant lag time while the computer looks for that routine.

Now, think through the same scenario with error trapping placed at the top. BASIC goes right to the top of the program, and encounters the error trapping routine on the second line it hits! There's virtually no time taken in searching for the error trapping routine, and hence it's almost impossible to break out of the program with RUN/STOP.

The TRAP 12 statement in line 12 may seem a little redundant, but it serves the same purpose. Commodore's manuals tell us there's no way to trap an error within the error trapping routine itself. But actually there is a way: you can make the error trapping routine turn itself on. It's why there are two RESUMEs at the end of the routine—one is to take care of the TRAP 12 statement at the beginning of the routine.

Just because the beginning of the routine is at the top doesn't mean you have to clutter up the beginnings of your programs with all the error tests and responses possible. Since error 30 (RUN/STOP) is the only one that's time dependent, we can safely tuck the rest of the error decision making down at the bottom.

A FINAL NOTE ON ERRORS

The best way to approach errors is to treat them as allies. Error codes can alert you to troubles on conditions that you'd never otherwise be able to test for or screen out. Used creatively, they can help you catch mistakes before they have a chance to do any real damage.

Chapter 12

Drawing Pictures

Make friends with a piece of graph paper. Graph paper is the only vehicle through which you can easily understand graphics on the Commodore 128. With a grid to look at, graphics layout becomes a breeze. Without a grid, it's a nightmare.

Let's start right off by drawing something using graphics screen one.

A SIMPLE GRAPHICS PROGRAM

While the C-128 offers several different graphic modes, the one we'll use most in our examples is graphic screen one, which is for *standard bit-map* graphics. In this mode, pictures are limited to one foreground and one background color for each *square* on the screen. You can use all 16 C-128 colors on graphics screen one, but the different colors cannot overlap within a single square. The advantage of this standard bit-map mode is that the pictures can be much more detailed. In the C-128's other mode, the multicolor bit-map mode, your pictures can squeeze more colors into small spaces, but the detail is somewhat compromised.

Graphics screen one is well suited for pictures of balloons, or other large-item portraits. The multicolor mode is best for colorful scenes, such as the cherry blossoms in a park at springtime.

Our program, listed in Fig. 12-1, draws a picture of a floppy disk. The picture has some artistic weaknesses, which we'll fix in a moment. The main thing is that it illustrates the use of several important graphics commands.

The first command is at line ten. GRAPHIC 1,1 selects graphic screen one and clears that screen of any previous pictures. The command GRAPHIC 1,0 would select the screen, but leave any previous drawings intact.

The GRAPHIC command also does something that is invisible, but could affect your programs if they're very, very large. The first time it's used, GRAPHIC helps itself to a 9K chunk of your program memory. Normally this isn't a problem, since your code would have to take up at least 48K to create any kind of graphics conflict. It is, however, a good idea to issue the GRAPHIC CLR command

```
0 goto 10
2 : disk program (1)
3 : by martin hardee
4 :
10 graphic 1,1
15 :
20 box ,90,50,160,120
25 :
30 circle ,125,85,10
35 :
40 circle ,125,99,2
50 circle ,125,116,2
55 :
60 draw ,127,99 to 127,116
70 draw ,122,116 to 122,99
75 :
80 getkey a$:graphic 0
```

Fig. 12-1. A simple program that draws a disk.

after you're done with graphics work, thereby freeing up some additional program memory.

Back to our program. Line 20 draws a box, with the upper left corner at coordinate 90,50 and the lower right corner at coordinate 160,120. You'll be able to see the computer draw the box right on the screen, though it's very fast. The beginning comma signals that the box should be drawn on the default screen, which is screen one.

The coordinates may be somewhat puzzling. Here's how they work: The coordinate 0,0 is at the upper left corner of the screen. Coordinate 319,199 is at the bottom right corner. It's a mirror reversal of the plotting you probably learned in school, since the lower coordinates are represented by higher (greater) Y values.

The drawing in Fig. 12-2, which was used in plotting this disk picture, may help clear things up. The coordinates are clearly marked.

Next comes the CIRCLE statement at line 30, which draws a circle around the coordinates 125,85. The radius is 10 pixels, giving the circle a diameter of 20 pixels. As you can see from the sketched drawing, this should be just about the size of the hole in the center disk.

Two additional CIRCLE commands place two small circles at the bottom of the disk; these will become the rounded corners of the disk's read/write slot.

Finally, two DRAW commands plot out parallel lines to connect the edges of the smaller circles. The DRAW command allows you to draw straight lines anywhere on the screen. Just specify the starting and ending coordinates, and the line will be drawn. DRAW can even be used to create polygons such as triangles and trapezoids. In fact, we could have even used it instead of the BOX command to draw the disk:

```
DRAW ,90,50 TO 160,50 TO 160,120 TO
90,120 TO 90,50
```

As with all other figure-plotting commands, a comma usually starts off DRAW's parameters, indicating that the default foreground color should be used.

The program ends up with a GETKEY line that switches back into text mode (GRAPHIC 0) when a key is pressed.

Improving the Program

When you run this program, you'll notice first that the disk appears to be too tall and narrow, and second, that it is devoid of any color or depth. There are simple commands to correct both of these situations.

First, we will remove the superfluous parts of the circles that form the edges of the disk read/write slot. We can do this because the circle command allows specification of an *arc*—part of the circle. Thus, if you wish to show only the top of a circle (270 degrees to 90 degrees) or the bottom of a circle (90 degrees to 270 degrees), you can do so with ease. So we'll add an extra comma and the appropriate degree measurements:

```
40 CIRCLE ,125,99,2,,270,90
50 CIRCLE ,125,116,2,,90,270
```

The additional comma, by the way, takes care of the circle's Y radius, which is normally the same as the X radius. We could also have simply included

another 2 at this place in the statement.

Now RUN the program. This disk looks better already, but not quite as good as it will.

Let's finish up with two commands. One will fill the disk with white, and the other will add a characteristic write-enable notch.

The first command is PAINT:

```
76 PAINT ,91,51
```

The PAINT command fills the boundaries of any figure with color. The coordinate given for PAINT may be anywhere inside the figure, but it may not be on a boundary line itself. If the coordinate *is* on a boundary line of the figure, PAINT will simply not work.

Now that you've seen how easily PAINT works, here's another secret: you can include a paint parameter in the BOX command. Normally, a box is drawn unfilled, as it was in this program, but by including an extra parameter at the end of the BOX command you can draw a solid box in the default color. You can even create voids by changing to the background color and then drawing a box:

```
78 BOX 0,155,60,160,63,,1 : REM NOTCH
```

The ,,1 at the end instructs the computer to fill this box as it is drawn. You will notice that the statement begins with a 0, which indicates that the box should be drawn in the background color instead of in the customary foreground color. Figure 12-3 shows a listing of the finished program, complete with an additional color command at line 5.

We'll be making some further enhancements to this program later in the chapter.

SOME NOTES ABOUT COLOR

Learning about the C-128's color schemes for the first time is a chore for even the most seasoned programmer. Here's why: there are seven differ-

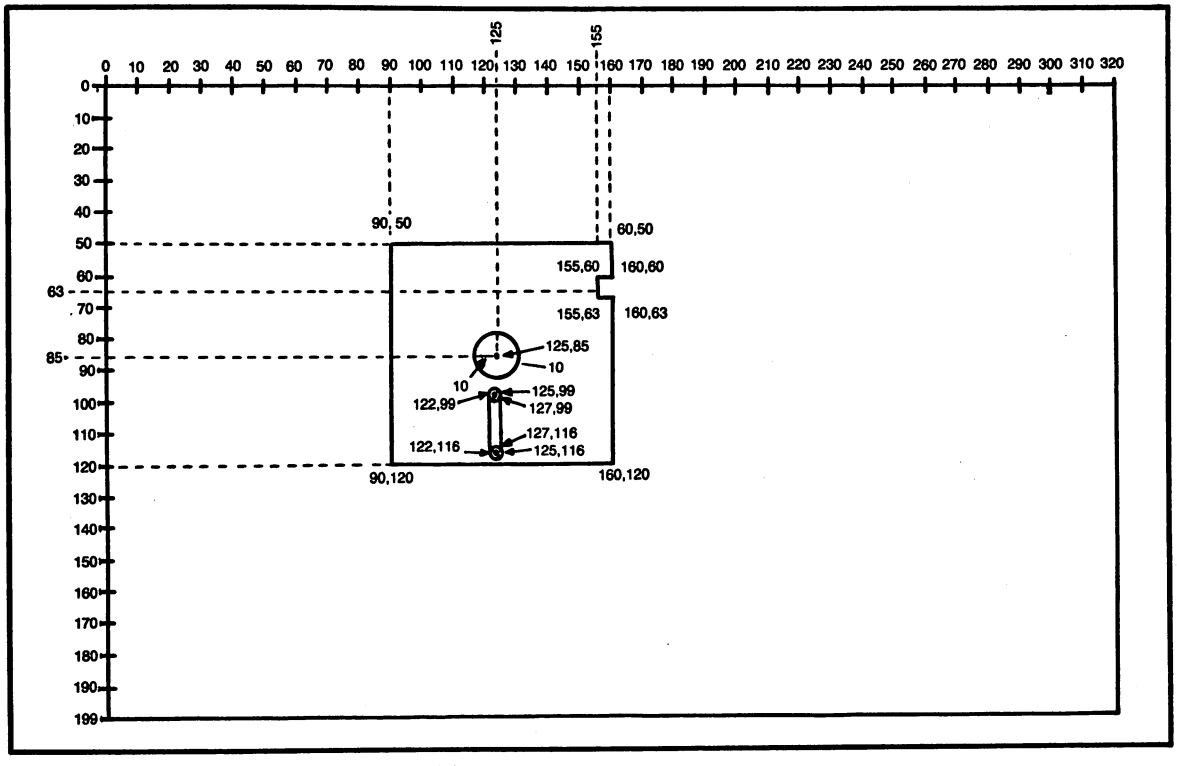


Fig. 12-2. Layout for a picture of a floppy disk.

```
Ø goto 5
2 : disk program (2)
3 : by martin hardee
4 :
5 color 1,2
1Ø graphic 1,1
15 :
2Ø box ,9Ø,5Ø,16Ø,12Ø
25 :
3Ø circle ,125,85,1Ø
35 :
4Ø circle ,125,99,2,,27Ø,9Ø
5Ø circle ,125,116,2,,9Ø,27Ø
55 :
6Ø draw ,127,99 to 127,116
7Ø draw ,122,116 to 122,99
75 :
76 paint ,91,51
78 box Ø,155,6Ø,16Ø,63,,1
8Ø getkey a$:graphic Ø
```

Fig. 12-3. An enhanced program that draws a disk.

ent color sources available, with 16 colors each. There are five different graphic *modes* (screens, if you will) onto which these colors can be displayed.

Graphic commands such as BOX and CIRCLE also include color source numbers (they're what you skip when you start these commands with a comma), but the numbers are completely different from those used for color source by the COLOR command!

The best axiom is always to know what screen you're on, what color you want to display, and whether it should be background or foreground. Mentally keeping track of these factors will go a good way toward soothing the confusion.

Usually you will pick a screen using the GRAPHIC command and you'll stay there. Normally, then, you'll be most concerned with the COLOR command and the screen it applies to. There are seven color sources available:

0 40-column text/bit-map graphics background
1 40-column bit-map graphics foreground
4 Border color for 40-column
5 40/80 text color

In GRAPHIC 1, the higher-resolution bit-map mode, the following color sources are available.

0 40-column text/bit-map graphics background
1 40-column bit-map graphics foreground
2 Multicolor graphics color #1
3 Multicolor graphics color #2
4 Border color for 40-column
5 40/80 text color
6 80-column background color.

In GRAPHIC 3, the multicolor mode, all color sources except for number six (80-column background) are available for use.

Note that while the background color for both text and graphics is set by 0 (for example, COLOR 0,1 for a black background), the text and graphics foreground colors are controlled by different numbers. The number 1 is used for graphics foreground, while 5 is used for text.

How does this affect the CHAR command, a creature that can function in both the graphics and text worlds? In the graphic mode, the CHAR command takes on the prevailing foreground graphics color. In the text mode, the CHAR command uses the text foreground color.

Finally, keep in mind that the same color rules applying to GRAPHIC 1 also cover GRAPHIC 2 (the split-screen mode). GRAPHIC 4, the split-screen mode for multicolor graphics follows the same general rules as does GRAPHIC 3. In fact, the only exception for graphic screens two and four is that the CHAR command cannot be used to display characters on the text portion of the screen in the split-screen mode. It just doesn't work, because the CHAR characters are written to a portion of the screen that isn't being shown. Keep in mind that CHAR *can* be used anywhere on graphics screens one and three.

Now that you've seen how colors work, you'll probably want to include some of your own color commands in the remaining programs from this chapter.

CHANGING THE SHAPE OF THINGS: SCALE

Our disk picture is far from complete. For one thing, it's too tall and skinny. Fixing the height and width of the disk is a function of the SCALE command. SCALE allows you to redefine the number of bit-map "pixels" on the X and Y scales.

In GRAPHIC 1 the normal setting is 320 for the X scale and 200 for the Y scale. Using the SCALE command, you can increase the number of points on both the X and the Y axes to 32,767.

Because SCALE only increases the number of available points, and can't be used to decrease it, the SCALE command always has the effect of making your picture smaller. How much smaller depends on how much bigger a scale you specify. For example, if you specified a scale of 640 for X and 400 for Y—doubling the normal settings—the resulting picture would be half as big.

Back to our disk picture: we will specify a scale that is just as wide (X = 320), but not quite so tall (Y = 250). Because SCALE should always be specified after the graphics screen has been selected, we'll place the SCALE command at line 12:

```
12 SCALE 1,320,250
```

The first parameter, 1, instructs the computer to turn scaling on. Because we're adding more points on the Y axis, the disk will appear more stout, less lean and tall.

Now, run the program. You can see there are a few more adjustments to make. Even though we've changed the scale of the drawing, the center hole still seems bigger than life, and the notch appears too small. All that's needed are some quick adjustments to the appropriate CIRCLE and BOX statements.

You'll remember that the CIRCLE command allows you to specify both an X and a Y radius. In the last example, we allowed the command to default the Y radius to the value already specified for X. But we can use a slightly smaller Y radius here to make the circle more round.

```
30 CIRCLE ,125,85,10,9
```

This command specifies that the Y radius should come up slightly short. The effect, under the current scale selected, will be a more perfect circle. You may want to experiment with other X and Y values to measure their effect.

Then there's the notch. It's too small. Rather than redraw the box, which is certainly an option, it would probably look nicer to have the notch drawn as part of the original disk drawing. To make this one final adjustment to the program, a more detailed draw command is required to replace the box command that starts the program. Figure 12-4 shows the program with this command. The program dispenses entirely with the BOX command at line 78.

You can change the SCALE command as often as you like. It won't affect drawings that are already displayed on the screen; it will only affect new ones that are drawn after the new scale goes into effect. This means you can use scale to make drawings from different sources more uniform in size.

MOVING YOUR PLOTTING POINTS

Once you've designed a drawing, it's a good idea to add a few variables that will help make it more portable. Specifically, you should have X and Y variables for each X and Y parameter in a picture. By placing your picture in a subroutine, and changing the X and Y values before the routine is called, you can move the drawing anywhere on the screen.

If you're planning to have portable pictures, it's best to change all the coordinates so the left corner of the drawing is at 0,0. If you don't take this approach, figuring out the relationships between various pictures can become a nightmare.

Figure 12-5 shows the first step in this process, where a value of 90 has been subtracted from all X coordinates, and a value of 50 has been taken from all Y coordinates. Notice that since the scale has remained the same, all radius and degree parameters in the CIRCLE statements have not been changed.

The next step is to add X and Y variables to

```
Ø goto 5
2 : disk program (2)
3 : by martin hardee
4 :
5 color 1,2
1Ø graphic 1,1
12 scale 1,32Ø,25Ø
15 :
2Ø draw ,9Ø,5Ø to 16Ø,5Ø to 16Ø,6Ø to 155,6Ø to 155,65 to 16Ø,65
   to 16Ø,12Ø to 9Ø,12Ø to 9Ø,5Ø
25 :
3Ø circle ,125,85,1Ø,9
35 :
4Ø circle ,125,99,2,,27Ø,9Ø
5Ø circle ,125,116,2,,9Ø,27Ø
55 :
6Ø draw ,127,99 to 127,116
7Ø draw ,122,116 to 122,99
75 :
76 paint ,91,51
8Ø getkey a$:graphic Ø
```

Fig. 12-4. An even better disk program.

```
Ø goto 5
2 : portable disk (1)
3 : by martin hardee
4 :
5 color 1,2
1Ø graphic 1,1
12 scale 1,32Ø,25Ø
15 :
2Ø draw ,Ø,Ø to 7Ø,Ø to 7Ø,1Ø to 65,1Ø to 65,15 to 7Ø,15 to 7Ø,7Ø
   to Ø,7Ø to Ø,Ø
25 :
3Ø circle ,35,35,1Ø,9
35 :
4Ø circle ,35,49,2,,27Ø,9Ø
5Ø circle ,35,66,2,,9Ø,27Ø
55 :
6Ø draw ,37,49 to 37,66
7Ø draw ,32,66 to 32,49
75 :
76 paint ,2,2
77 color 1,12
8Ø getkey a$:graphic Ø
```

Fig. 12-5. A program that draws disks anywhere.

```
Ø goto 5
2 : portable disk (2)
3 : by martin hardee
4 :
5 color 1,2: rem foreground white
6 color Ø,1: rem background black
1Ø graphic 1,1
12 scale 1,32Ø,25Ø
15 :
17 ctr=3 :rem first color
2Ø for x=1 to 245 step 8Ø
25 :
3Ø :   for y=1 to 175 step 8Ø
35 :
37 :       ctr=ctr+1
38 :       color 1,ctr
4Ø :       gosub 1ØØØ :rem draw disk
45 :
5Ø :   next
55 :
6Ø next
65 :
7Ø getkey a$:graphic Ø
75 :
8Ø end
85 :
86 :
1ØØØ draw ,x+Ø,y+Ø to x+7Ø,y+Ø to x+7Ø,y+1Ø to x+65,y+1Ø to
     x+65,y+15 to x+7Ø,y+15 to x+7Ø,y+7Ø to x+Ø,y+7Ø to x+Ø,y+Ø
1Ø1Ø :
1Ø2Ø circle ,x+35,y+35,1Ø,9
1Ø3Ø :
1Ø4Ø circle ,x+35,y+49,2,,27Ø,9Ø
1Ø5Ø circle ,x+35,y+66,2,,9Ø,27Ø
1Ø6Ø :
1Ø7Ø draw ,x+37,y+49 to x+37,y+66
1Ø8Ø draw ,x+32,y+66 to x+32,y+49
1Ø9Ø :
11ØØ paint ,x+2,y+2
112Ø return
```

Fig. 12-6. An improved program for relocatable disks.

these values, as shown in Fig. 12-6. In this listing, the picture has been turned into a subroutine. For purposes of example, a FOR . . . NEXT loop has been added that will draw a set of twelve disks on the screen. A CTR variable is included to change the color of each disk. Since the program is written for the bit-map mode, the disks have been separated slightly so that the colors from adjacent

character blocks don't conflict with one another. If you want to view this color overlap, change the STEP factor to 75. Otherwise, sit back and enjoy the color.

You can see from this simple example how versatile the Commodore 128's graphics commands can be. You'll probably want to use the last program to experiment further with SCALE and other commands.

In the next chapter you'll learn all about true animation, using manipulable graphic objects called *sprites*.

Chapter 13

Animation

Animation is one of the best things going on the Commodore 128. C-128 BASIC has been designed to handle almost everything, from drawing the shapes to be animated, to moving them around, to accommodating collisions and display conflicts.

In this chapter, you'll see how to create a simple video game that includes a spaceship, monsters, missiles that can be fired, and explosions. All of these effects will be achieved through computerized pictures known as *sprites*. Sprites are the cornerstone of computer animation.

WHAT IS A SPRITE?

Sprites are a very nimble, manipulable kind of figure that can be drawn very fast. Sprites are also known on some computers as *shape tables*.

To fully appreciate what sprites do, it's necessary to think about what life would be like without them. In Chapter 12 you saw how a picture of a disk or other object can be drawn at any place on the screen. What would it take to move the disk around the screen? Primarily, you'd be drawing the disk over and over again at adjacent spots. But to avoid simply creating a long white streak across the screen, each new move of the disk would require that you also erase the area where the disk had previously been displayed. Not only would it be tedious to program; it would be slow, because the disk would have to be redrawn with each move. If you ever wanted to remove the disk from view, you would have to redraw it at its last position in the background color.

Sprites do all of this work with two simple commands. What's more, once you've started a C-128 sprite moving, your program can progress to other things. Movement is automatic, requiring no loops, repositioning, or other tricks.

Perhaps the best thing about sprites is that they're so easy to create. A picture that would require fifteen or twenty lines of code using BOX, CIRCLE and DRAW statements can be designed and saved on disk in a minute or two using the Commodore 128's versatile *sprite definition* (SPRDEF) command.

This is not to say that there aren't some draw-

backs. There are limits to the size of a sprite: even in an "enlargement" mode, it won't take up more than approximately a ten percent section of the screen. Sprites containing circles also can appear more jagged then figures plotted using the CIRCLE command, because sprites are comprised of miniature squares.

CREATING YOUR OWN SPRITES

SPRDEF is one of the niftiest commands available on the Commodore 128. As soon as you type SPRDEF (no parameters), the screen will clear and a large design window will fill the left side. Your pictures will be drawn inside this window using the cursor keys and a few simple commands.

Up to eight sprites may be designed in this manner. Because you can store each set of sprites in a separate disk file, the number of sprites that can be created and displayed is unlimited.

Creating a Monster or a Spaceship

Once you've invoked SPRDEF, and the SPRITE NUMBER? prompt appears, you are ready to begin. Type 1 and press RETURN.

Next, press SHIFT-CLR/HOME to clear the sprite of any electronic debris that may have been in memory. Be certain there is only one plus sign at the upper left corner of the screen. If there are two plus signs, you're in the multicolor mode and you should press the M key, since the examples in this chapter use the higher-resolution bit-map graphics mode.

You may also want to set the foreground color in which you'll be working, although this color won't affect the color of the sprite within our programs. You can change the working color by pressing the C= key along with the number keys 1 through 8.

Drawing a sprite is simply a matter of positioning the cursor at the right place, and pressing the 2 key. To erase a block, simply position the cursor at the desired square and press the 1 key. The 1 and 2 keys are normally in the *automatic* mode, which means that when you draw or erase a block, the cursor moves one position to the right.

The first thing to do is experiment. Place the cursor about halfway down on the left side and draw a straight line. As you plot the line using the 2 key, a smaller version of the picture will appear on the right side of the screen. This is the actual sprite as it will appear in your programs. Drawing vertical lines is a little more tedious, since you have to plot the squares using the arrow keys.

SPRDEF includes two commands that can enlarge the sprite for easier viewing. Pressing X makes the sprite wider. Pressing Y makes it taller. These keys work as toggles, so pressing them again will return the sprite to its original size. It's generally best to work in the enlargement mode because you can see what's going on better.

Now, let's save the sprite on disk, by exiting SPRDEF and issuing a BSAVE command:

- Press SHIFT-RETURN to save the sprite in memory.
- Press RETURN at the SPRITE NUMBER? prompt.

Sprites are stored in a special area of memory in bank zero. The command to save them is always the same:

```
BSAVE "LINE SPRITE",B0,P3854 TO P4096
```

Note that you can save on disk sprites under any legal filename name you wish. The command to retrieve sprites looks like this:

```
BLOAD "LINE SPRITE"
```

The BLOAD command contains no additional parameters, because the computer can determine the previous bank and memory locations from information stored in the file. Since sprites are always stored in the same area of memory, they're always saved on disk using the same parameters, and you may BLOAD them without specifying any extra parameters.

GETTING DOWN TO WORK

Figures 13-1, 13-2 and 13-3 contain three

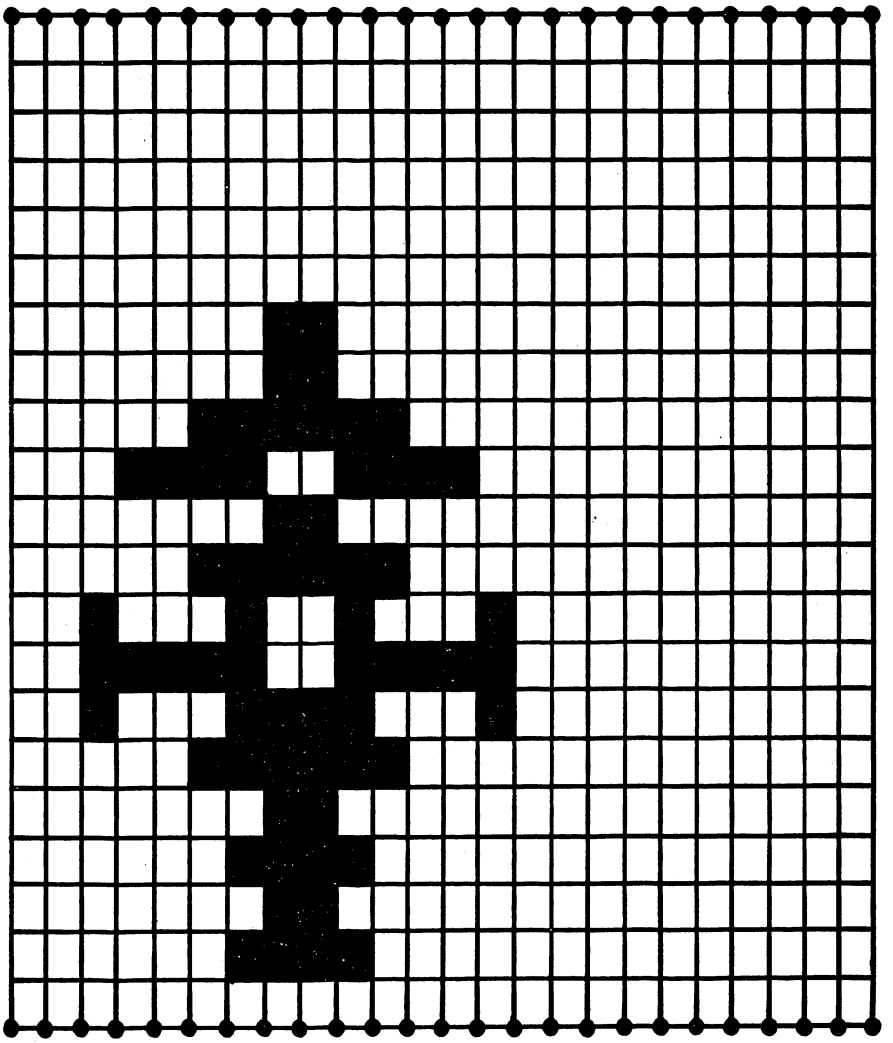


Fig. 13-1. A spaceship sprite. This sprite may be recreated using SPRDEF.

shapes that will be used in the video game later in this chapter. Draw and save them according to the following steps:

1. Enter SPRDEF.
2. Select the correct sprite number for each shape, plot it out, and save it in memory using SHIFT-RETURN. Then, continue with the next sprite.
3. When all three shapes have been designed and saved in memory, exit SPRDEF by pressing RETURN.
4. Save the sprites with the command:

```
BSAVE "TEST SPRITE",B0,P3854 TO P4096
```

Be certain that you draw all three sprites, and that you specify the correct number. Also make sure you have BSAVEd the sprites under the name TEST SPRITE. This is the filename used in the example program.

ANIMATING SPRITES: A VIDEO GAME EXAMPLE

The job of displaying and animating sprites

falls on the shoulders of two commands: SPRITE and MOVSPR.

The SPRITE command turns on the display of a sprite, sets the sprites color, and determines what happens when two sprites vie for the same spot on the screen. It may also be used to enlarge a sprite, just as you did with the X and Y keys when using SPRDEF.

The MOVSPR command is used not only to move a sprite, but also to position it on the screen. There are two modes to MOVSPR. In one mode, the command is used simply to place a nonmoving sprite at a particular place on the screen. In the other mode, MOVSPR actually moves the sprite at a specified angle and speed. The sprite won't stop moving or change velocity until it encounters another MOVSPR command. When using this second mode, it's generally a good idea to position the sprite ahead of time with the first MOVSPR mode. If you issue the second type of MOVSPR without first positioning the sprite, you'll never be sure just where the sprite will start its path on the screen.

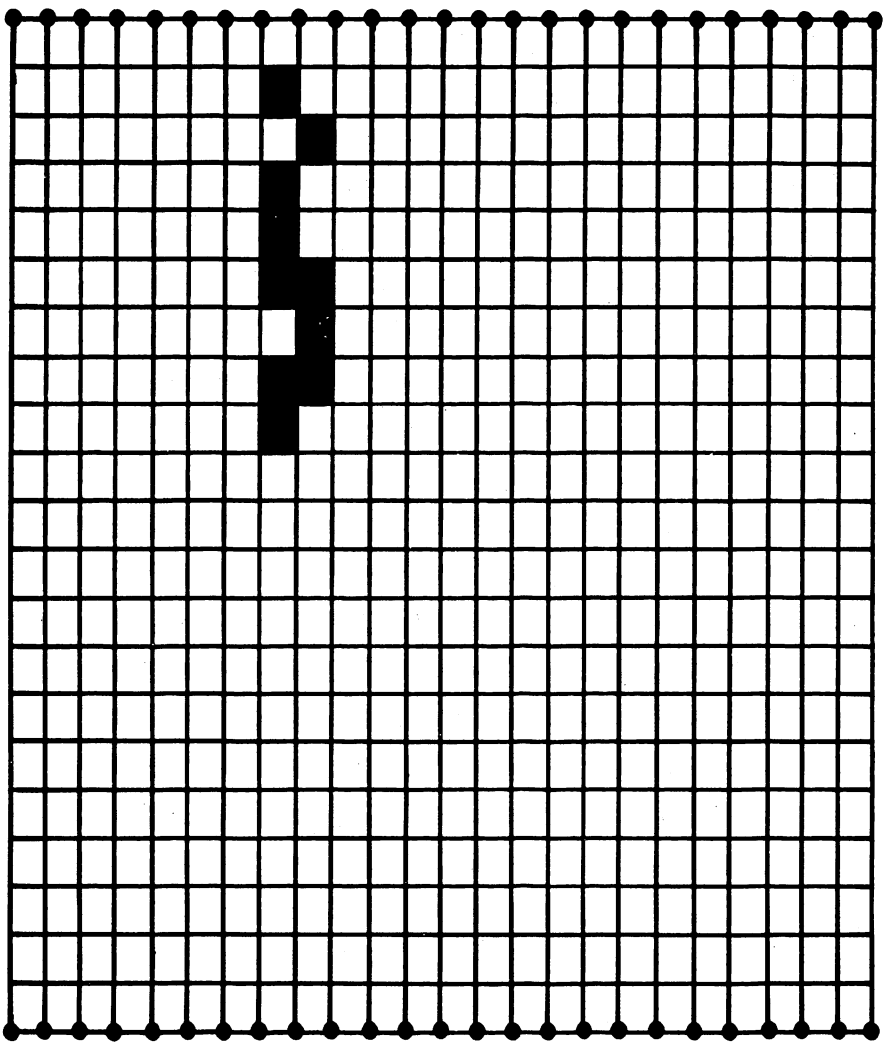


Fig. 13-2. A missile sprite.

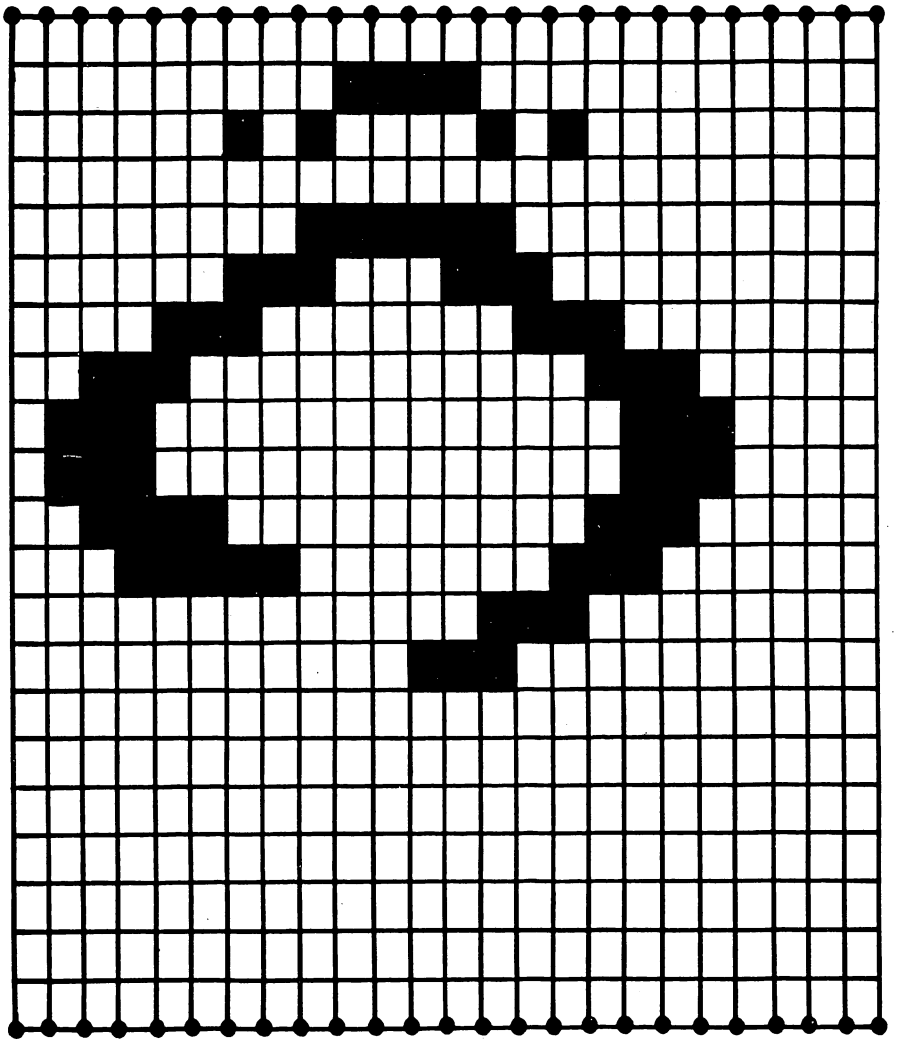


Fig. 13-3. A monster sprite.

Using SPRITE and MOVSPR

SPRITE and MOVSPR can be used at any time, on any screen in the 40-column mode. You can even display and move sprites directly over printed words or program listings when in the GRAPHIC 0 (text) mode.

To turn on a sprite, simply issue the SPRITE command. The following turns on sprite 3, the monster you created:

```
SPRITE 3,1
```

There's no predicting where the sprite will appear on the screen. If you have not used MOVSPR since last powering up the computer, it's likely that the sprite will be out of viewing range, and won't be displayed at all. We'll fix that with MOVSPR:

```
MOVSPR 3,200,60
```

The sprite should appear at the upper left portion of your screen. (If a blob appears instead, issue the BLOAD "TEST SPRITE" command and then type the MOVSPR line again).

Now that the sprite is positioned, you can move it. Here's the command to move sprite 3 at a 90

degree angle (left to right) and at top speed:

```
MOVSPR 3,90 #15
```

This command uses the second mode of MOVSPR you read about earlier. The sprite will keep moving until you issue another MOVSPR command with an angle of zero, hit RESTORE-RUN/STOP, or turn off the computer. The sprite will keep moving at the same relative angle even if you reposition it on the screen with the first *static* mode of MOVSPR.

Untangling Sprite Coordinates

It's important to note that sprites work under a different X and Y plotting scheme than you'll find on any of the C-128's graphic screens. For example, you'd expect the upper left corner to have a coordinate of 0,0—just like GRAPHIC 1,2,3 and 4. Instead, the upper left sprite coordinate is 24,50. You can plot a sprite at 0,0; it simply won't show up.

The last visible coordinate for a sprite (lower right corner) is 344,250.

Here's the good news: No matter what 40-column or graphics screen you're on, your sprites will look the same and will maintain the same scale. The only place sprites don't work is the 80-column mode.

A Program Example

You now know enough about sprites to write your own simple video game. We'll start with a program that simply moves sprites around the screen.

This program will do the following:

- Move a spaceship from left to right at the bottom of the screen.
- Put the monster in motion, moving in the opposite direction across the top of the screen.
- Fire a missile from the spaceship when the space bar is pressed.

The listing in Fig. 13-4 does all of this using the SPRITE commands you already know. Let's look at how it works.

Line 10 starts off by making sure the correct sprites are in memory; it loads the binary TEST SPRITE file. Any sprites currently in memory will be overwritten with the new sprites from this file.

Next, the program switches to GRAPHIC 1, sets a testing variable for a space, and sets a speed variable at line 70. The way the program is designed, changing this variable will correspondingly increase or decrease the speed of the ship and the monster.

The statement at line 80 uses the full complement of options available with the sprite command:

```
SPRITE 1,1,2,0,1,1,0
```

In English, the line reads:

Select the spaceship (sprite 1).
Turn it on (so it's visible).
Select white as the color (color 2).
Give the sprite top priority (it will obscure other sprites that come into contact with it).
Expand it in a horizontal direction (X expand = 1— on).
Expand it in a vertical direction (Y expand = 1— on).
The sprite is standard (0 = not multicolor).

It's helpful to work with the manual when you're typing in new SPRITE lines, since there are so many parameters.

Line 90 positions the spaceship at the bottom of the screen, and line 110 starts it moving from left to right.

Lines 120, 130, and 140 turn on the monster, position it at the top part of the screen, and starts it moving from right to left.

Line 150 starts a short loop that controls movement of the spaceship and checks to see if the space bar was pressed.

You're probably wondering why you'd need a routine to move the spaceship, since a MOVSPR command previously set it in motion. It comes down to this: when we fire the missile, we'll want it to come directly from the nose of the spaceship. Unfortunately, there's no way to know the exact po-

```
2 :
3 : mercury's invaders (1)
4 : by martin hardee
5 :
6 :
7 :
10 bload "test sprite"
30 graphic 1,1
45 :
46 :
60 t$=chr$(32) :rem for spacebar test
70 adj=4       :rem speed
75 :
76 :
80 sprite 1,1,2,0,1,1,0  :rem turn on ship
90 movspr 1,200,150      : rem position ship
95 :
110 :  movspr 1,90 # adj  :rem start ship moving
120 :  sprite 3,1,8,0,1,1 :rem turn on monster
130 :  movspr 3,200,60    :rem position monster
140 :  movspr 3,270 # adj :rem start monster moving
150 do
210 :      x=rsppos(1,0):y=rsppos(1,1)
220 :      movspr 1,x+adj,y
240 :      get a$:if a$=t$ then gosub 340:rem fire missle
260 loop
265 :
330 end
335 :
340 : rem shoot
350 sprite 2,1,2,1,1,1    :rem turn on missle
360 movspr 2,x+adj+5,y    :rem position
370 movspr 2,360 # 15     :rem launch
400 return
```

Fig. 13-4. The Mercury 1 program: a simple video game.

sition of a moving sprite. The best we can do is keep a leash on our moving spaceship by having it hop across the screen. Because the sprite is already in motion, you won't notice these hops; the sprite is moving at just the right speed to catch up to the next hopping point just as the command in line 220 repositions it there.

The two RSPPOS functions on line 210 return the previous hopping point of the sprite. RSPPOS returns the last sprite position set using the first mode of MOVSPR—the static mode where a sprite issimply positioned on the screen. This function comes in handy when you're moving different sprites every which way and want to keep track of where they are. The first parameter refers to the sprite number. The second refers to the axis (0 for X and 1 for Y).

Line 240 performs the "fire a missile" subroutine whenever the space bar is pressed.

The program keeps running until you press

STOP or RESTORE-STOP. RESTORE-STOP is preferred because it also turns all of the shapes off and returns you to the graphics screen.

Shoot 'em Up

The routine at line 340 takes advantage of the MOVSPR command's two different modes. Line 350 turns on the missile sprite. Line 360 positions the sprite at the nose of the space craft by figuring out where the ship is currently positioned. It does this by taking the x position of the last hop (X), adding ADJ (which should reflect approximately how far the spaceship has moved since the space bar was pressed) and adds five as another adjustment.

The missile is launched at line 370, where it moves at an angle of 360 (upward) at top speed (#15).

The first time you fire a missile with this program, you'll notice minor design deficiencies. First, the missile never stops; it keeps up its bottom-to-top trajectory until the space bar is pressed again, and the missile sprite is repositioned somewhere else. Second, the missile is a dud. When it hits the monster, it simply keeps going. There's no explosion; there's no score.

The final version of the program will correct these flaws.

Bumps and Collisions

In order to make something blow up or disappear when it hits the border of the screen, the computer must know when the paths of two objects have crossed. The Commodore 128 includes two features to help you do this: BUMP and COLLISION.

The BUMP function simply tells if any sprite has collided with an object on the screen. The number returned by BUMP(1) will indicate if any sprites have collided. The number returned by BUMP(2) indicates if the sprite has collided with text or some other nonsprite entity on the screen.

To determine *which* sprite collided, you have to do some arithmetic. The number returned by the BUMP function indicates two raised to the power of the product of the sprite number *minus one*. If two sprites were involved in the collision, two is raised to the power of each sprite number minus one separately. The two results are then added together. Here's how you would calculate the number for a collision between sprites 3 and 4 and turn the sprites off if they had collided. The extra subtraction has been added to stress the fact that you must subtract one from the sprite number before raising 2 to the power of it:

```
BOOM= (2↑(3-1)) + (2↑(4-1)
IF BUMP(1)= BOOM THEN SPRITE
3,0:SPRITE 4,0
```

BUMP can be a fickle function that doesn't always behave as you'd expect, especially if there are several sprites moving around at once. You will definitely have to experiment.

The program in Fig. 13-5 (Mercury 2) solves our first "missile won't stop" problem by drawing a line at the top of the screen with the CHAR command (the underline is created by a repetition of the C= and U). When the missile hits this border, BUMP(2) will be equal to 2. Lines 190 and 230 turn off the sprite and stop it if this condition occurs.

The COLLISION command at line 20 performs a similar function, forcing the computer to jump to a subroutine whenever two sprites collide. COLLISION tests for collisions are like TRAP tests for errors. The first parameter indicates the type of collision to be alert for (1 = sprite-to sprite, 2 = sprite-to-text, 3 = light pen). If a collision has occurred, the program is sent to the subroutine at the specified line number.

Since the missile and monster are the only two likely collision candidates in this program, we can safely create an explosion sound and remove the two sprites from the screen whenever there's a sprite-to-sprite collision.

Sound, Fuel and Ammo

Several other new features have been added to this program to make it more realistic. For example, when you fire a missile, the computer plays a quick blast of notes. The amount of ammunition, set at the top of the program is also decremented.

New explosion sounds, pointed out earlier, have

```
2 :
3 : mercury's invaders
4 : by martin hardee
5 :
6 :
7 :
10 bload "test sprite"
30 graphic 1,1
40 char ,0,0,""
45 :
46 :
50 brder=2:fuel=2000:ammo=80
60 t$=chr$(32) :rem for spacebar test
70 adj=4       :rem speed
75 :
76 :
80 sprite 1,1,2,0,1,1,0  :rem turn on ship
90 movspr 1,200,150      : rem position ship
95 :
100 do while fuel>0 and ammo>0
110 :  movspr 1,90 # adj  :rem start ship moving
120 :  sprite 3,1,8,0,1,1 :rem turn on monster
130 :  movspr 3,200,60    :rem position monster
140 :  movspr 3,270 # adj :rem start monster moving
150 :  bang=0
160 :  do until bang or fuel <0 or ammo<0
170 :     fuel=fuel-(5*adj)
180 :     gosub 510       :rem show score
190 :     if bump(2)=brder then sprite 2,0
200 :     collision 1,410 :rem a hit!
210 :     x=rsppos(1,0):y=rsppos(1,1)
220 :     movspr 1,x+adj,y
230 :     if bump(2)=brder then sprite 2,0
240 :     get a$:if a$=t$ then gosub 340:rem fire missle
250 :  loop
260 loop
265 :
270 char ,15,10,"game over"
280 sprite 1,0:sprite 2,0:sprite 3,0
285 :
290 do until a$=chr$(27) : rem escape
300 :  getkey a$
310 loop
320 graphic 0
330 end
335 :
```

Fig. 13-5. A more complex video game.

```
340 : rem shoot
350 sprite 2,1,2,1,1,1    :rem turn on missle
360 movspr 2,x+adj+5,y    :rem position
370 movspr 2,360 # 15     :rem launch
380 tempo 250:play "v1t0o5 v2t0o6 v3t0o6 v1c v2e v3b v1e v2c v3f"
    :rem sound
390 ammo=ammo-1           :rem less ammo
400 return
401 :
410 rem                        explosion
415 :
420 sprite 3,0            :rem turn off monster
430 sprite 2,0            :rem turn off ship
435 :
440 sound 1,700,10,1,0,1,3,304
450 sound 2,1000,10,2,400,10,3,100
460 tempo 40:play "v3t3ocwc"
465 :
470 bang=1:tally=tally+1:adj=adj+1
480 if adj>15 then adj=15
485 :
490 gosub 510
500 return
505 :
506 :
510 rem                        show score
515 char ,2,22,"score: "+str$(tally)+"0"+"  fuel: "+
    str$(fuel)+"   ammo: "+str$(ammo)+"   "
520 return
```

been added in the explosion routine.

Finally, within the main loop of the program, a score subroutine is continually called to show the remaining fuel and ammunition, and to let you keep track of how many points you have.

Other Changes You Can Make

To make the explosion even more realistic, you could design two or three additional sprites that are only displayed for a split second upon impact. These sprites could mimic the spectacle of an explosion, as the fire quickly spreads and then diminishes.

You might also want to test for arrow keys, and change the angle of the spaceship accordingly. An angle of 90 would be set if the right arrow key were pressed. An angle of 270 would be set for the left arrow.

Finally, you could use SPRDEF's copy (C) command to copy the monster to several additional sprite blocks, thereby allowing several monsters on the screen, running at different speeds and angles. You'd also have to add several BUMP tests in the explosion routine, so you'd know *which* monster has been blown up.

Graphics are limited only by how much imagination you have, and how much time you're willing to spend on programming. The first part is up to you. Fortunately, the Commodore 128 makes the programming part easy.

Chapter 14

Music and Sound

Lots of folks have looked longingly at computerized music, studied what's involved, and quickly given up. After all, even your Commodore 128 manual begins its chapter on sound with a dissertation on frequencies, sine waves, fundamentals, harmonics, and sound envelopes. It would seem you have to be an accomplished musician and physicist rolled into one.

Wanna know a secret? Making beautiful music on the C-128 really isn't so complicated. You don't need an engineering degree. In fact, playing a simple song on your Commodore is as easy as picking out a tune on that little xylophone you had when you were a kid. Producing three-part harmony is just like humming along to chopsticks on a friend's piano.

MUSIC THE EASY WAY

Forget for the moment about ADSR, harmonics, square waves, frequencies and everything else. Here's all you need:

PLAY

This one word can produce a wide, richer range of sounds than any command available on almost any other microcomputer. Here's some of what PLAY can do:

- It can play notes just as if you were touching the keys to a piano.
- It can switch from baritone to tenor to soprano in an instant. With a range of five octaves, PLAY sweeps the range of the human voice.
- It can change its timbre to mimic the sounds of ten different musical instruments.
- It can sing in chorus using three different voices at once,or it can play solo.

PLAY is a deep fog horn in the distant night. It's the good-time sound of a honky-tonk piano banging out a familiar ballad. It's a trumpet heralding the start of a race. It's a xylophone playing so fast that you can barely pick out the notes.

Admittedly, the Commodore 128 doesn't always produce music that completely matches these

instruments or moods, but with the tricks you'll learn in this chapter, you'll be able to create an astounding variety of sound.

SOME QUICK NOTES ON MUSIC

Let's start with what we all know: Music is made up of notes. The order of notes determines the melody, while the length of the individual notes helps to determine whether it's a waltz or a samba. Tempo defines the speed at which the music is played . . . and that's about it. While there is a great deal to the study of music, the basics are pretty simple.

The Commodore 128 sticks faithfully to those basics. For example, musicians have invented a concept called *time* to help them measure the length of the notes they're playing. You've undoubtedly heard of 4/4 time, or 3/4 time, or 6/8 time. You will find no time command on the Commodore 128, because the computer doesn't need it. Time is simply a measuring device to help humans compose and play their music.

The Commodore is only concerned with the length of the notes you give it to play. The time of the music will simply fall into place, just as it does as notes are played on any other instrument.

While it's always helpful if you can read sheet music, the Commodore's PLAY command is really designed for those of us who can't. If you can sing in the shower, you've probably got enough musical talent to get started on the Commodore 128.

PLAYING INDIVIDUAL NOTES

Here's one of the first things you probably played on your toy xylophone or baby piano. It's a partial scale:

```
PLAY "CDEFG"
```

That set of notes will probably sound like it came from an electric piano. Producing a recognizable tune is a matter of telling the C-128 what notes to play. Here's the beginning of Mary Had a Little Lamb:

```
PLAY "AGFGAAA"
```

The PLAY statement is not limited to just a few notes. In fact, you can specify up to 160 characters in a play command. Here's a rendition of the song in its entirety:

```
PLAY "AGFGAAAGGGAAA AGF GAA
AAGGAGF"
```

When you issue this command on the C-128, you'll notice something right away: the music will sound flat and uninspired. That's mainly because we're not changing octaves to take advantage of higher notes, and because we're not pausing between clauses of the music. Here's a slightly improved version:

```
PLAY "O4 AGFGAAA R GGG R A O5 CC
O4 R AGFGAAAAGGAGF"
```

Even though this line adds only two new features, there's a big difference in the way it sounds. First, you will notice a new letter, R, which tells the computer to *rest* for a note. The R accounts for the pause in play. Second, we have defined the an octave at the beginning of the PLAY statement. An octave is simply used to define a certain level of highness or lowness for the notes being played. Octave 4, defined here by O4, is toward the upper end of the musical spectrum on the C-128, and is the machine's default octave.

Our PLAY statement also uses the O parameter twice more—once to switch to octave 5 (and play two notes), and once more to switch back to octave 4. Had we not switched up to octave 5—but still entered the two C notes—the melody would dip down to a lower C in octave 4, which wouldn't be the desired result at all. Each new octave starts at C, just as on a piano.

The Black Keys

Even if you've never studied music, you've no doubt heard about sharps and flats. These are the black keys on the piano, that represent a note that's halfway in between two natural keys. For example, D sharp is one-half a tonal step between D and E.

The term sharp is used to describe a note that's halfway above a natural note, and the term flat is used for a note that's a half-step below. Therefore, D sharp and E flat are really the same note.

On the Commodore 128, a pound sign (#) precedes notes to be played as sharps, and a dollar sign ($) precedes notes to be played as flats. In the PLAY statement, D sharp would be represented like this: #D. The note E flat would be played like this: $E. Again, since both notes are really the same, they'd sound identical.

Here's the beginning of Skip to M' Lou, played in the key of G. It includes an F sharp:

```
PLAY "O4 BBGGBB O5 D R O4 AA #F#F
AA O5 C"
```

Sharps and flats are very important. If you take the sharp sign away, you'll notice a dramatic change in the tonality of the song. It will sound like someone is hitting the wrong notes.

PLAYING IN HARMONY

Because it has three separate voices, your C-128 can do a good bit more than simply pick out simple tunes; it can provide background accompaniment to any melody.

One example of how harmony can be used is a chord. Here's a C cord produced using the V (voice) parameter:

```
PLAY "V1 C V2 E V3 G"
```

This chord sounds much fuller than any single note could. The notes here, played simultaneously, complement each other.

The V parameter is used to select voices, and should precede any other information for the voice, such as an octave setting or notes. Perhaps the simplest form of harmony is simply to play the same note simultaneously at different octaves. Here's an example of three C's played simultaneously:

```
PLAY "V1O4 C V2O5 C V3O6 C"
```

You've probably noted that parameters in all the PLAY statements so far have been broken up by spaces. This is only to make the statements more readable. A statement such as the following would work just as well, but it is nearly impossible to decipher:

```
PLAY "V1O4CV2O5CV3O6C"
```

Thus for the most part, you'll see the extra white space inserted into musical program lines. Later in this chapter, we'll talk about other ways to make the PLAY statement easier to use and comprehend.

Back to harmony: one of the best known examples of two-note harmony is the song "Chopsticks"—known on the piano by every precocious seven year old. Figure 14-1 shows how the beginning of "Chopsticks" would be played by the Commodore 128. It's just as spine shattering as the version played on a baby grand.

The program starts off with a never-ending DO . . . LOOP that keeps the song playing in perpetuity. You can break out of the song by hitting the RUN/STOP key.

Line 20 sets the TEMPO at which the music is played. Although the moderate tempo of seventy specified here is best for a song like Chopsticks, you could specify a lower speed (down to a tempo of one), or a faster speed (up to a tempo of 255). At a tempo of 255, TEMPO speeds along at tickertape speed. If you change the TEMPO setting to one, the pause between notes will be an unbearable 30 seconds (you can escape from this torture by pressing RESTORE and RUN/STOP simultaneously).

Since the main body of Chopsticks is played twice in the song, this melody has been placed in a subroutine, which starts at line 150.

The first statement in this routine uses the PLAY command to set voice one at octave four and voice two at octave five. There are no notes in this PLAY statement; it's only used to set the *default* octaves for the voices.

The next three lines contain FOR . . . NEXT loops that play each set of notes six times. The PLAY statement in line 170 starts by playing F un-

```
10 do
20 tempo 70
30 gosub 150
40 play "v1o5 c v2o6 c r w m"
50 play "v1o5 c v2o6 chr wm"
60 play "v1o5 c v2o6 chr wm"
70 play "v1o4 b v2o5 dhr wm"
80 play "v1 e v2 ahr wm"
90 gosub 150
100 play "v1c v2o6c hrw r m"
110 play "v1o5 f v2o5 ghr wm"
120 play "v1o5 c v2o6c hrw m"
130 print"chopsticks!"
140 sleep 1:loop
150 rem  : main subroutine
160 play"v1o4 v2o5"
170 for x=1to6:play "v1 f v2 g hr wm":next
180 for x=1to6:play "v1 e v2 g hr wm":next
190 for x=1to6:play "v1 d v2 b hr wm":next
200 return
```

Fig. 14-1. Chopsticks.

der voice one and G under voice two. Because the notes are under different voices, they'll be played simultaneously, producing harmony. The next parameter, HR, sets the note duration to a half note, and then plays a rest, pausing for half a measure to provide separation between the notes. If you removed the HR, the notes would seem to play faster and would run together.

The last section, WM, resets voice 2 for whole notes (a full four beats) and instructs the computer to finish playing the current line before moving on. If the M parameter were not included, voice 1 would begin playing the next note while voice 2 was still resting. The M parameter allows you to place all voices on hold while the current notes or rests finish playing. It's a convenient alternative to specifying separate rests for each voice.

Once the routine is finished, the program continues at line 40 by playing a two-octave C chord and resting for one full measure before continuing.

The main body of the tune is called again at line 90, followed by a three-note finale. At the end of the song, the word "CHOPSTICKS!" is displayed on the screen, the computer pauses for one second (thanks to the SLEEP command) and then LOOPs back to the top, where it begins the never-ending cycle again.

WHOLE NOTES, HALF NOTES

Up until now, most of the examples you've seen have used "whole" notes: a single note per measure (in 4/4 time). Whole notes are the default on the Commodore 128. In a few examples, we've included half note rests and other devices that change the timing of the notes being played. Figure 14-2, the chorus from the song Buffalo Gals, employs notes of varying durations. The program starts off with two sixteenth notes (IGG), followed by two quarter notes (Q] E], followed by two sixteenth notes (IDD). Figure 14-3 shows a complete chart of note durations available on the Commodore 128, along with their sheet music counterparts.

DEFINING INSTRUMENTS

The Commodore 128 comes with 10 instruments built in. They are the piano, accordion, calliope, drum, flute, guitar, harpsichord, organ,

```
10 play "o5 igg q$g e idd qe idd o4 hb"
20 play "o5 qd icc o4 ha"
30 play "o5 qe idd o4 hb"
40 play "o5 igg q$g e idd qe idd o4 bb"
50 play "o5 qd idd qc o4 iaa"
60 play "wg"
```

Fig. 14-2. Buffalo Gals.

trumpet, and xylophone. You may define an instrument either inside a PLAY statement or through the ENVELOPE command, which also allows you to redefine the sound of a synthesized instrument. Each voice can be assigned a different instrument, and the assignments can even change from note to note if you desire.

The easiest way to set an instrument is through the T parameter of the PLAY command. For example, this statement would play the note C using the Commodore 128's organ synthesizer:

```
PLAY "T7 C"
```

You can create chords for a single instrument by including additional notes and voices:

```
PLAY  "V1T7 C V2T7 E V3T7 G"
```

By assigning different instruments to each voice, you can create musical accompaniment:

```
PLAY  "V1T7 C V2T4 C V3T5 C"
```

The statement above plays the note C as an organ in voice one, a flute in voice two, and a guitar in voice three.

By fiddling with the speed, you can even redefine the feeling of the sound. For example, the following statement creates a fog horn by stretching the notes played by the organ synthesizer:

```
TEMPO 1:PLAY "V1O1T7 C V2O2T7 E
V3O3T7 G"
```

The program in Fig. 14-4 illustrates the complete range of musical instruments available, by placing the chorus to "Buffalo Gals" in a loop. With each pass through the loop, the value of X is increased, the name of the new instrument is read from a data statement, and the song is played with a new instrument. The new lines are 2, 5, 6, 7, 65, 70, 80 and 100.

VOLUME

The VOL command can be placed anywhere in a program to turn sound up or down. A command of VOL 0 turns the sound completely off. You can also affect sound volume through the U parameter

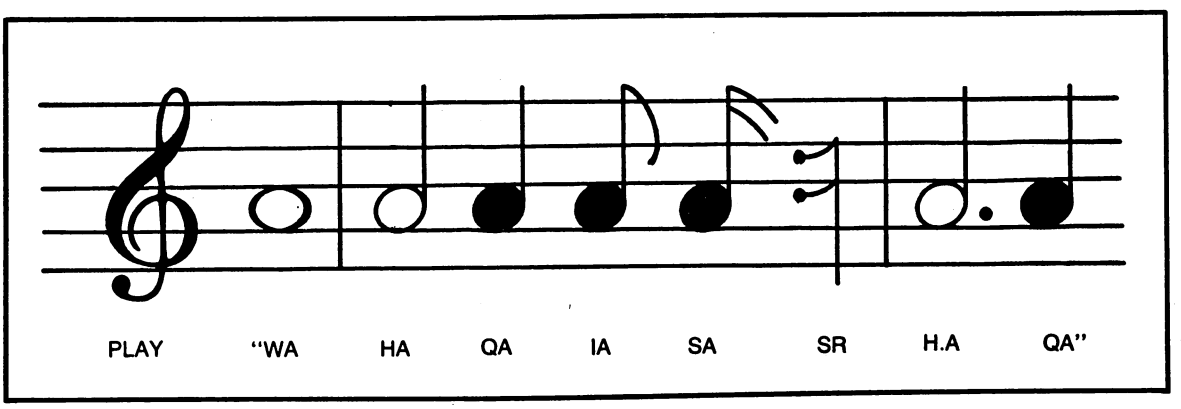


Fig. 14-3. Sheet music notes and durations.

```
2 scnclr:tempo 2Ø
5 for x=Ø to 9
6 :    read a$:printa$
7 :    play "t"+str$(x)
1Ø :   play "o5 igg q$g e idd qe idd o4 hb"
2Ø :   play "o5 qd icc o4 ha"
3Ø :   play "o5 qe idd o4 hb"
4Ø :   play "o5 igg q$g e idd qe idd o4 bb"
5Ø :   play "o5 qd idd qc o4 iaa"
6Ø :   play "wg"
65 :   sleep 1
7Ø next
8Ø end
1ØØ data piano,accordion,calliope,drum,flute,guitar,harpsichord,organ,
    trumpet,xylophone
```

Fig. 14-4. A program to play different instruments.

of the PLAY command:

PLAY "U5 C"

When volume parameters are placed in close proximity they tend to cancel one another out. Therefore, to change the volume between successive notes, insert a rest between the two.

PLAY "U5 C" : PLAY "R U15 C"

The volume command does not discriminate between voices. If you turn the volume down for one voice, the other two voices are dampened to the same degree.

MORE ON HARMONY

The more complicated the music you're playing, the more tricks you'll need to make it come off right. Figure 14-5 is a menu-driven program that uses the multiple-voice features of the C-128 to their fullest. The theory of the program is simple: small "chunks" of notes are stashed in separate data statements. Typically, these "chunks" are equivalent to one measure, although their real purpose is to make it easier to write music that harmonizes. The longer notes (generally the *base line* of the song) are placed in the first data statement to ensure that they're played first. It's important to do this, so that the long notes begin to play in one voice, giving the shorter notes in another voice a chance to catch up. If there is no second voice, an extra comma is inserted between the first and third voices.

The beginning envelope setting is a modification of the default organ envelope. By lengthening the attack, release, and sustain times, and changing the wave width from 512 to 4000, we've achieved a much deeper sound from our newly defined organ. The only way to really understand envelopes is to experiment with them! The envelopes for all default instruments are listed in your manual and can be similarly modified to obtain interesting results. If you get stuck with an instrument you don't like, you can reset it to the default parameters, or simply hit RESTORE and RUN/STOP to restore the machine to its normal mode.

Here's basically how the program works, once a selection is chosen through the menu in the 60000s. First, the program resets the variables V1$, V2$, and V3$, just as a precaution. Next a DO loop that will continue until the program hits a ## in the data statements is started. For each line, the notes are read from data statements into V1$, V2$, and V3$. They're then played at line 220, with all voices set to our newly defined instrument envelope 0. The V1$ variable is played by voice 1, V2$ is played by voice 2, and V3$ is played by voice 3.

```
5 tempo 40: envelope 0,0,10,10,10,2,4000
6 gosub 60000
7 end
200 v1$="":v2$="":v3$=""
205 do while v1$<>"##"
210 read v1$,v2$,v3$
220 play "v1t0"+v1$+"v2t0"+v2$+"v3t0"+v3$
250 loop
255 return
260 end
9999 rem          : vivaldi
10000 data" 1"
10005 data"q o3 f $b",,"o1h $b"
10010 data"o3q a $b f ",,"o1h f "
10020 data"o3q d f $b ",,"o1h $b "
10030 data"o3q a $b f ",,"o1h f "
10040 data"o3q d f ",," "
10042 data"o4 c ",,"o2h c "
10044 data"o3q $b o4 c o3 f ",,"o1h f "
10046 data"o3q a o4 c d",,"o1h f "
10048 data"o4q c d o3 w$b",,"o1w $b"
10050 data ##,##,##
10099 rem          : call to post
10100 data" 2"
10105 data "","","o5q.c f am"
10110 data "o2 wc ","o4 wc ","o6 h c c irq c cm"
10120 data "o1 wa","o4 wf","o5 h a a irq a am"
10130 data "o1 wf","o4 wa","o5 h f a fm"
10135 data "o1 wc","o5 wc","o5 c hrr"
10160 data "","","o5q.c f a"
10165 data "o2 wc ","o4 wc","o6 h c c irq c cm"
10170 data "o1 wa","o4 wd","o5 h a a irq a am"
10180 data "o1 w c","o4 we","o5 h c c cm"
10190 data "o1 w f","o4 wf","o5 h. f"
10195 data ##,##,##
10199 rem            shave & haircut
10200 data" 3"
10205 data o1wf,,o4wf
10210 data o1wc,,o4hcc
10220 data o1wd,,o4wd
10222 data o1we,,o4wc
10225 data r,,r
10230 data o1wc,,o4we
10240 data o1wf,,o4wf
10250 data ##,##,##
10299 rem            boogie
```

Fig. 14-5. An electronic jukebox. Select the tune of your choice.

```
10300 data " 4"
10302 data o5wc,o3irh.b,o1hc   :rem start at c
10304 data ,,o1he
10306 data o5wc,o4irh.c,o1hg
10308 data ,,o1ha
10310 data ,,o1h$b
10312 data ,,o1ha
10314 data ,,o1hg
10316 data ,,o1he
10318 data o5wc,o4irh.e,o1hc
10320 data ,,o1he
10322 data o5wc,o4irh.e,o1hg
10324 data ,,o1ha
10326 data ,,o1h$b
10328 data ,,o1ha
10330 data ,,o1hg
10332 data ,,o1he
10334 data o5wf,o4irh.f,o1hf  :rem up to f
10336 data ,,o1ha
10338 data ,,o2hc
10340 data o5wf,o4irh.f,o2hd
10342 data ,,o2h$e
10344 data ,,o2hd
10346 data ,,o2hc
10348 data ,,o1ha
10350 data o5wc,o4irh.c,o1hc  :rem back to c
10352 data ,,o1he
10354 data o5wc,o4irh.c,o1hg
10356 data ,,o1ha
10360 data ,,o1h$b
10362 data ,,o1ha
10364 data ,,o1hg
10366 data ,,o1he
10372 data o5wg,o4irh.g,o1hg  :rem up to g
10374 data ,,o1hb
10376 data o5wg,o4irh.b,o2hd
10378 data ,,o2he
10380 data ,,o2hf
10382 data ,,o2he
10384 data ,,o2hd
10386 data ,,o1hb
10388 data o5wf,o4irh.f,o1hg
10390 data ,,o1hb
10392 data o5wf,o4irh.a,o1hd
10394 data ,,o1he
10396 data ,,o1hf
10398 data ,,o1he
10400 data ,,o1hd
```

```
10410 data ,,o1hb
10420 data o5wc,o4irh.g,o1hc   :rem back to c
10422 data ,,o1he
10424 data o5wc,o4irh.e,o1hg
10426 data ,,o1ha
10428 data ,,o1h$b
10430 data ,,o1ha
10432 data ,,o1hg
10434 data ,,o1he
10436 data o5wc,o4wc,o1hc
10498 data ##,##,##
60000 rem menu
60005 out=0:do until out
60010 restore:scnclr
60020 char ,10,5,"1. vivaldi"
60030 char ,10,6,"2. call to post"
60040 char ,10,7,"3. shave & a haircut"
60045 char ,10,8,"4. boogie"
60075 a=100:do while a>4 and not out
60080 getkey a$:a=val(a$):ifa=0 then out=1
60090 loop
60092 if out then exit
60095 ctr=0:do until a$=str$(a)
61000 :  ctr=ctr+1:read a$:char ,0,22,"["+str$(ctr)+"]",1
61002 char ,0,23,"scanning . . ."
61005 loop
61006 char ,0,23,"playing . . . "
61010 gosub 200
61020 loop
```

The menu routine, by the way, selects the appropriate data statement based on the number chosen, by scanning the data statements for that number.

To exit the program, simply press anything other than 1,2,3, or 4. If you'd like to add other tunes, simply follow the same data statement format (number at the beginning, ## at the end) and change line 60075 to allow for more options. Don't forget to add the song title to the menu.

A BRAND NEW INSTRUMENT

By combining the power of the C-128's ENVELOPE commands with the versatility of its different voices, it's possible to create a complete new instrument. The program listed in Fig. 14-6 is short but immensely powerful; it allows you to play music on the bottom two rows of the C-128 keyboard and switch octaves using the numbers keys 1 through 5. The sound is something like a cross between a big church pipe organ and a concert piano with the third pedal engaged. You'll have to hear it to fully appreciate how it sounds.

The way it works is simple. After defining a tempo and envelope that will apply to each note played, the program begins a loop that continues until ESC is pressed.

At line 90, a counter is continually bumped and reset with each pass. It will always have the values 1, 2, or 3, corresponding to one of the C-128's three musical voices.

```
Ø :           goto 1Ø :rem skip rem's
1 : keymusic
2 : by martin hardee
3 : o   = octave selected
4 : a$  = note keyed/converted for play
5 : adj = adjustment for octave
6 : ctr = counter for current voice
7 : out = exit flag
8 :
9 :
1Ø scnclr
4Ø :
5Ø rem define new tempo and envelope
55 :
6Ø tempo 1Ø: envelope Ø,Ø,9,8 ,1Ø,2,1536
7Ø o=3
75 :
8Ø do until out
9Ø :  ctr=ctr+1: if ctr>3 then ctr=1
1ØØ : gosub 15Ø  : rem get note
1Ø5 :
11Ø : play "v"+str$(ctr)+"tØ"+"o"+str$(o+adj)+a$
12Ø : print a$" ";
13Ø loop
135 :
136 :
14Ø end
145 :
15Ø getkey a$    :rem get note
155 :
16Ø rem conversion table
165 :
17Ø on instr("zsxdcvgbhnjm,l.:/",a$) goto 21Ø,22Ø,23Ø,24Ø,25Ø,
    26Ø,27Ø,28Ø,29Ø,3ØØ,31Ø,32Ø,33Ø,34Ø,35Ø,36Ø,37Ø,38Ø
18Ø if instr("12345",a$) then o=val(a$):a$="":return :rem set octaves
    with close#'s
19Ø if a$=chr$(27) then out=1:a$="" :return: rem escape to end
2ØØ a$="":return
2Ø5 :
21Ø a$="c":adj=Ø:return
22Ø a$="#c":adj=Ø:return
23Ø a$="d": adj=Ø:return
24Ø a$="#d":adj=Ø:return
25Ø a$="e": adj=Ø:return
26Ø a$="f": adj=Ø:return
27Ø a$="#f":adj=Ø:return
28Ø a$="g": adj=Ø:return
```

Fig. 14-6. Keymusic: a new electronic instrument.

```
290 a$="#g":adj=0:return
300 a$="a": adj=0:return
310 a$="#a":adj=0:return
320 a$="b": adj=0:return
330 a$="c": adj=1:return
340 a$="#c":adj=1:return
350 a$="d": adj=1:return
360 a$="#d":adj=1:return
370 a$="e": adj=1:return
380 a$="f": adj=1:return
```

The routine starting at line 150 does a number of things:

- It gets a keystroke
- It branches to one of many routines if one of the keys on the bottom two rows was pressed
- It sets a musical note if one of these keys was pressed
- If escape was pressed, it sets the OUT flag

The heart of the routine is the ON . . . GOTO, which branches to statements based on the position of A$ within the INSTR function at line 170. If Z was pressed, the program branches to 210, where a note of C is set. If an S was pressed, the program branches to 220 where a note of C sharp is set, and so on. Why didn't we just use a long sequence of IF statements? Because this ON . . . GOTO approach with INSTR is much, much faster.

Once the note has been set, it is played in the current voice, which is determined by the value of CTR (1, 2, or 3). With each keystroke, a different voice is selected, lending an echo-like sound to the music. The multiple-voice approach also ensures that notes won't get backed up or be skipped because you happen to be pressing the keys faster than the computer can convert and play the notes. If you press keys in quick succession, you can even form chords in this manner (X, C, and B played in rapid succession create a pleasant sounding C triad).

If you hold keys down for any time, the note will repeat and feed into itself, sounding something like a mandolin.

The higher octaves, obtained by pressing the 4 or 5 keys, sound like bells. The lower octaves, set by pressing 1 or 2, are more akin to a majestic pipe organ.

In time, you may even want to experiment with the envelope statement to lend different shades of sound to this new instrument we've invented.

SOME FINAL NOTES

There's one last note when using the C-128's music commands: the C-128 is slightly out of tune! If it seems as though the Commodore notes are off pitch from a piano or accordion or some other instrument that's supposed to stay in tune, don't blame the instrument. The C-128 is approximately 1 1/2 steps off from perfect pitch.

The Commodore 128 can provide you with lots of good musical entertainment and learning. If you enjoy working with the machine's musical commands, you may want to consider investing in some sheet music that can be transcribed into PLAY statements. It will be reproduced as faithfully as if you were running a player piano.

If you're interested in explosions, phaser shots, and other such sound effects, you'll also want to experiment with the SOUND command, which is covered extensively in your C-128 manual.

Appendix A

A Refresher Course in BASIC

Perhaps it has been a while since you've worked with the BASIC computer language, or perhaps this is your first time using BASIC. Whatever your reason for reading this section, you'll find Commodore BASIC 7.0 to be a comfortable, easy-to-understand, easy-to-learn language.

HOW BASIC WORKS

You've probably already looked in your *Commodore 128 User's Guide,* which details how simple calculations and text can be displayed on the screen. But there's much more to BASIC than simple computations.

Once you're comfortable with the language, you can use it to sort data, file information, search for data, calculate your household budget, and perform tasks you've probably never even thought of attempting. The key to BASIC's operations is line numbers. While most commands can be typed into the computer and executed immediately, in order to store commands for later operation, they must be preceded by a line number. For example, the following statement prints the word HELLO on the screen:

```
PRINT "HELLO"
```

If you want to make this part of a sequence of events, which happens only when the program is run, it must be preceded by a line number:

```
10 PRINT "HELLO"
```

A series of such lines constitutes a program. Program statements are generally executed in order. For instance, a statement on line 10 would be acted on before a statement on line 11 would be:

```
10 PRINT "HELLO"
11 PRINT "HOW"
12 PRINT "ARE"
13 PRINT "YOU"
```

Note that it isn't necessary to type lines in order. You could enter line 12 before lines 10 and 11. The

computer will automatically place the program lines in order (as each new line is added).

Nor is it necessary to place line numbers one apart. In fact, program line numbers are usually spaced 10 or 20 apart:

```
10 PRINT "HELLO"
20 PRINT "HOW"
30 PRINT "ARE"
40 PRINT "YOU"
```

Leaving program lines some elbow room allows space for other line numbers to be inserted later in the same.

TAKING A NEW DIRECTION

Normally the computer takes program lines one at a time and in order. There are several BASIC commands, however, that allow you to skip over certain lines or redirect the computer's attention to a completely different section of the program. In fact, it's a good idea—once you're comfortable with simple programming—to set up special *routines* that the computer performs separately from the program as a whole. Here's an example:

```
10 PRINT "HELLO"
20 GOSUB 200
30 PRINT "GOODBYE"
40 END
200 PRINT "I'M GLAD TO MEET YOU"
210 RETURN
```

In this example, a subroutine prints the words "I'm glad to meet you." Programmers refer to the operation at line 20 as a *call*. "Line 20 calls the routine at line 200," you'll hear them say. The RETURN command at line instructs the computer to return to its previous place in the program (just as if it had kept a book mark at line 20). Because the routine is called between lines 10 and 30, the sentence at line 200 will appear on the screen after the word HELLO and before the word GOODBYE:

```
HELLO
I'M GLAD TO MEET YOU
GOODBYE
```

The advantage of using this routine is that it can be called several times during the program, without having to repeat line 200. For example, you could put a GOSUB 200 before line 10, and after line 30 (say, at lines 5 and 35). With the additional calls, the screen display would look like this when the program is run:

```
I'M GLAD TO MEET YOU
HELLO
I'M GLAD TO MEET YOU
GOODBYE
I'M GLAD TO MEET YOU
```

Subroutines are especially useful when you plan to repeat complicated or bulky program functions several times during a program. They make your program easier to follow and less bulky, since often-used routines don't have to be repeated. In this book we'll be using subroutines extensively for keyboard entry, filing, sorting, and display applications.

Another Way to Redirect Program Control

Another command, GOTO, is also used to redirect the computer to a different line. In this example, GOTO is used to skip over line 30:

```
10 PRINT "HELLO THERE"
20 GOTO 40
30 PRINT "THIS WILL NEVER APPEAR"
40 PRINT "GOODBYE"
```

True to its promise, the statement in line 30 will never show up on the screen.

Naturally, this type of program structure serves no useful purpose as it stands, but when used with decision-making operations (such as we'll discuss later in this section), GOTO becomes substantially more dynamic.

BASIC AS A SECOND LANGUAGE

Here are some BASIC commands in procedures that you will likely encounter during your first experiences with the new language.

Print

You've already seen several examples of the PRINT command and probably realize by now that it is used to display information. If you've peeked at your Commodore 128 User's Guide, you may also have discovered that the question mark (?) can be used as shorthand for typing PRINT each time you want to use the command.

The PRINT command has several forms, all of which print information onto the screen, to a printer, or into a file. PRINT may be used to display numbers, text, and calculations—or a combination of them:

```
PRINT "HELLO"
PRINT 2
PRINT 2 * 5
```

In the last example, the asterisk (*) is used as a multiplication symbol.

Any text to be displayed must be enclosed in quotes, but text and numbers are easily blended into the same PRINT statement

```
PRINT "HELLO";2 * 5
```

Text must be enclosed in quotes because groups of letters (such as HELLO) are interpreted as *variables* when they aren't inside quotes. When HELLO is placed in quotes, it will appear as a word.

The quick, simple application of variables is one of BASIC's key talents. BASIC allows you to determine values much as is done in an algebra formula. This makes BASIC amazingly flexible, because variables may represent different values at different times, even in the same formula.

An easy example to picture is a simple household budget. Part of the formula for determining the amount of money saved each month might be:

```
SAVD = PAY - XPENSES
```

In this example, the computer figures the value of the variable SAVED based on the stated formula. If the amount of PAY or XPENSES changes, the value of SAVD will likewise be adjusted. Another type of variable, called a *string* variable because it contains strings of characters, is used to store both text and numbers.

```
NAME$ = "JAN"
```

Whenever a string variable such as NAME$ is printed in a program, the characters contained in that string will be displayed. Thus these lines:

```
10 NAME$="JAN"
15 PRINT NAME$
```

would display:

```
JAN
```

Naturally, if NAME$ is assigned a different value (collection of characters) somewhere down the road, that different value (say, "MIKE" or "CARLA") would be displayed instead.

There are some limits to BASIC's versatility. You can't, for instance, perform the same types of mathematical operations with string variables as with numeric ones, but there are ways to add, or *concatenate,* string variables together (such as adding a first and last name together), and we cover these methods extensively in the book.

Using Variables to Their Fullest

BASIC can do much more than simply assigning values within a program; it can let you or other users determine the values while the program is actually chugging along. This feature gives your variables and formulas truly unlimited flexibility.

To reassign a variable's value while the program is running, we need a way to accept entry from the keyboard and store it in the variable. We have all of this in the INPUT command, which accepts input from the keyboard or from a disk file. If you wanted to find out a user's name, and store that name in the NAME$ variable, the following line would do the trick:

```
10 INPUT NAME$
```

This is the simplest form of the INPUT com-

mand, and as such it has drawbacks. First of all, the user will only know that the computer is requesting an entry by a single question mark that appears on the screen. There would be no clue as to what the program really wants. Can you think of a way around this problem? One way would be to add a print statement before line 10, giving the program operator some instructions:

```
 5 PRINT "PLEASE ENTER YOUR
   NAME";
10 INPUT NAME$
```

The semicolon following the PRINT line is very important. It signals the computer to let the cursor remain where it is, instead of advancing a line. Since the cursor points to the position of the next character to be displayed, the next character typed will appear to the right of "PLEASE ENTER YOUR NAME." Without the semicolon, the next character printed (which would be a ? in this case) would appear on the next line. But the semicolon ensures that the ? will be placed at the end of the sentence. Here's what it would look like when the program is run:

```
PLEASE ENTER YOUR NAME?
```

The flashing cursor would be positioned to the right of the question mark, and the response would be printed to the right of the prompt and question mark as the characters are typed from the keyboard.

Another way to accomplish the same sort of thing is to combine the "PLEASE ENTER YOUR NAME" prompt within the input line itself:

```
10 INPUT "PLEASE ENTER YOUR
   NAME"; NAME$
```

As you can see, this approach combines a function of PRINT within INPUT, and keeps both operations together.

The same trick can be used with numeric variables. Here's a short program that puts all of these concepts to work:

```
10 INPUT "PLEASE ENTER YOUR
   NAME";NAME$
20 INPUT "PLEASE ENTER YOUR
   PAY";PAY
30 INPUT "PLEASE ENTER YOUR EX-
   PENSES";XPENSES
40 SAVD = PAY - XPENSES
50 PRINT "YOUR SAVINGS ARE: ";
   SAVD
```

Naturally, there are many more things we could do with this listing. For example, we could instruct the computer to print a special message if there was no money saved (SAVD = 0), or to print a different message if savings were negative (SAVD is less than zero).

Words We're Not Allowed to Use

You'll notice that the SAVD variable does not contain the letter E, because the word SAVE in BASIC is *reserved* for a special computer operation (saving information). BASIC won't allow you to use such reserved words for variable, and will generally produce a ?SYNTAX ERROR or some other computer equivalent to the Bronx cheer. XPENSES follows the same rule, because EXP is a reserved word in C-128 BASIC.

Reserved words include all BASIC commands. In short, don't assign a variable the same name as a BASIC command. It simply confuses the computer and causes an error.

Decisions, Decisions

Close to the top of the list of serviceable BASIC commands is the IF function. Let's say you want to print a message, such as the ones we discussed above. If the amount saved equals zero, you might say "NO MONEY HAS BEEN SAVED." The statement would look like this:

```
42 IF SAVD=0 THEN PRINT "NO
   MONEY HAS BEEN SAVED"
```

The above statement instructs the computer to do two things:

1. It tests to see if SAVD has a value of zero.
2. If SAVD equals zero, it prints the message that follows the test.

If SAVD does not equal zero, the program skips to the next statement without printing anything. Thus, the only time this message appears on the screen will be when no money has been saved. How about another statement—one that will handle negative SAVD values (in other words, a statement for handling budget deficits). If you'll think back to your last brush with classroom mathematics, you'll remember that the less than sign looks like this <. So we can do another test that's very similar:

```
44 IF SAVD < 0 THEN PRINT "THERE
   IS A BUDGET DEFICIT"
```

(If you were a politician, and this were a government budget, you could change the message to read: "THERE IS A REVENUE SHORTFALL")

There's one more step you might want to take to make the program really radiate. Remember the line that prints the amount saved? We could block that line so it isn't displayed on the screen if either of these two conditions exists (SAVD=0 or SAVD<). Logically, the only time the SAVD value should be displayed is when SAVD is greater than zero—when there was actually money saved:

```
50 IF SAVD > 0 THEN PRINT "YOUR
   SAVINGS ARE: "; SAVD
```

Now, the value in SAVD is only printed if a certain condition is satisfied (SAVD is greater than zero).

There are other combinations of these tests that are also handy to know. When you wish to test for a variable that is NOT EQUAL to a given number, a less than and greater than sign are used together:

```
52 IF SAVD < > 0 THEN PRINT "SAVD
ISN'T EQUAL TO ZERO"
```

In math, of course, the symbol is simply an equal sign with a slash through it. Computers are not equipped with all the symbols mathematicians have available to them. If you wish to test for something that is "greater than or equal to" or "less than or equal to", again you can combine the two signs together, in any order:

```
54 IF SAVD = > 0 THEN PRINT "SAVD
   IS LESS THAN OR EQUAL TO 0"
56 IF SAVD < = 0 THEN PRINT "SAVD
   IS GREATER THAN OR EQUAL TO 0"
```

Such decision-making is one of the real provinces of computers. During this book, we'll talk about how to make the most of decision-making tests, and how to use them as efficiently as possible.

On and On . . .

Another way to let the computer make decisions is by using the ON GOTO and ON GOSUB commands, which examine a particular variable and branch to a list of line numbers based on the value of the variable. Here's an example:

```
40 ON A GOSUB 100,200,300,400
```

In the program containing this line, the variable A would probably hold the value of a user's selection from a menu. In this line, if A=1 then the computer performs the subroutine at line 100; if A=2 the subroutine at 200 is performed, and so on.

The lines don't have to be in order; the following variation is perfectly acceptable:

```
40 ON A GOSUB 200,300,100,400
```

Line 40 would perform the subroutine at line 200 if A=1, the subroutine at 300 if A=2, the subroutine at 100 if A=3, and the subroutine at 400 if A=4.

If A is greater than the number of choices, or is equal to zero, the computer does nothing; it simply continues on to the next line.

FOR . . . NEXT

Another feature you'll be using quite a bit is

something programmers call *recursion*, and which the rest of us call *doing something over and over again*. BASIC's built-in FOR . . . NEXT structure allows operations to be repeated as many times as necessary. Once you get used to its slightly unusual syntax, you'll find that FOR . . . NEXT is very easy to work with.

Imagine for a moment a track and field event in which you are instructed to run around a track a certain number of times. At the beginning of the event, you're given a handful of marbles, with each marble representing one lap. One of these marbles is thrown to the ground at the beginning of your run. Each time you pass the starting point, you toss another marble on the ground. When you have thrown the last marble, the game is over.

The FOR . . . NEXT loop functions in the same fashion as our athletic event. At the beginning of the loop, you specify how many times the operation should occur (in essence, how many marbles you're starting with). The computer will perform the set of operations within the loop until it marks off the specified number of items (until it runs out of marbles). BASIC's NEXT command is used to designate the bottom of the loop, and to go to the next run of the loop (the equivalent of throwing down a marble). Here's an example of a FOR . . . NEXT loop that prints HELLO fifteen times:

```
10 FOR COUNTER = 1 TO 15
20 PRINT "HELLO"
30 NEXT
```

How would you print HELLO thirty times? Simply replace the 15 (in line 10) with 30.

Notice that NEXT defines the bottom of the loop, and that the variable named COUNTER is used to count off the successions of the loop. Line 30 could also have read:

```
30 NEXT COUNTER
```

because COUNTER is the variable being decremented, but the computer takes it for granted that counter is the variable being used, since counter was the last variable defined with a FOR statement. Therefore the statement is generally abbreviated with a simple NEXT.

The COUNTER variable can be printed, added to, subtracted from, and generally treated like any other numeric variable. Sometimes, programs print the loop variable to indicate the loop's progress. We've done so here using the variable name MARBLES (yes, you can use any variable you wish—not only COUNTER):

```
10 FOR MARBLES = 1 TO 10
20 PRINT "MARBLES NOW ON
   GROUND="; MARBLES
30 NEXT
```

Try this out on your Commodore 128, and you'll see that it prints the number of marbles as the computer goes through each pass of the loop.

It's also interesting to note that you can play around with the variable being used to count successions of a loop. You can add to or subtract from the current value of the counter to make the loop skip over or repeat certain operations. Try experimenting on your computer with the following listing:

```
10 FOR MARBLES = 1 TO 10
15 IF MARBLES = 3 THEN MARBLES =
   5
20 PRINT "MARBLES NOW ON
   GROUND="; MARBLES
30 NEXT
```

In this example, the fourth progression through the loop would be skipped entirely. You can change the values in line 15 to produce all kinds of different results. If you make MARBLES equal to or greater than a value of 10, the value will be printed on the screen, but the loop will end. That's because line 10 in effect instructs the computer to perform the loop only while the value of MARBLES is "less than or equal to ten" and "greater than or equal to one." If you think about it, setting the MARBLES counter to 10, 11 or some higher number is a good way to exit the loop prematurely.

Backward Counting and Other Tricks. You may have already noticed that the previous ex-

ample counts only the marbles on the ground, but there are times when you want to count down instead of counting upward. Is there a way to count the marbles still remaining in the runner's hand? Actually, there are two ways. One would be to simply subtract the MARBLES counter variable from 10 at each pass, as in line 25 here:

```
10 FOR MARBLES = 1 TO 10
20 PRINT "MARBLES NOW ON
   GROUND="; MARBLES
25 PRINT "MARBLES STILL IN
   HAND=";10-MARBLES
30 NEXT
```

As the value of MARBLES counts upward, the value in (10 – MARBLES) decreases. If the formula seems a little confusing, work through a few examples.

There's another way to step backward through a loop, and this second way is less complicated than the above example, which uses a forward counting loop and then subtracts. Here's what a backwards counting loop looks like within a program:

```
10 FOR MARBLES = 10 TO 1 STEP -1
20 PRINT "MARBLES STILL IN
   HAND=";MARBLES
30 NEXT
```

The only real difference is that the loop starts at ten and works its way down to one. The STEP –1 statement is very important because it tells the computer to step backward each time. Without STEP, the computer would perform the loop only once.

You may want to experiment further with STEP as you gain familiarity with BASIC, because it can also be used to skip forward or backward by increments. The following loop, for example, counts marbles by twos:

```
10 FOR MARBLES = 1 to 10 STEP 2
20 PRINT "MARBLES COUNTED=";
   MARBLES
30 NEXT
```

If you're not yet comfortable with FOR . . . NEXT, take a few minutes to experiment with it on your Commodore 128. As you begin to use the computer, you'll see that FOR . . . NEXT is one of BASIC's most splendid structures.

Creating Remarkable Programs

Remarks, or REMs, are perhaps one of the least utilized yet most important features of BASIC. REMs enable you to place remarks anywhere within a program. Only you and other programmers will see these notes; the operator will be oblivious to them.

REMs can tell you a great deal about a program listing, where it came from, and how it works. For example, it's always a good idea to place the date, time and program name at the top of the program. If a question ever arises as to what program is in memory, and when it was last updated, you'll know by simply LISTing it:

```
 5 REM    ==============
10 REM    ADDRESS LIST PROGRAM
20 REM    BY PATTY PROGRAMMER
30 REM    LAST UPDATE 23 JUL 85
40 REM          TIME: 9:00 P
50 REM    ==============
60 REM    COPYRIGHT MCMLXXXV
          BY PATTY PROGRAMMER
70 REM    ALL RIGHTS RESERVED
```

Again, none of these lines would be executed by the computer—they're only remarks to the programmer. The meat of the program would actually start below these beginning lines.

REMs are also extensively employed at the end of program lines to clearly identify an operation. Here is a variation on one of our previous programs, clarified with a few REMs:

```
10 FOR MARBLES = 1 TO 10: REM
   START AT 1 AND LOOP TO 10
20 PRINT MARBLES:  REM PRINT
   CURRENT #
30 NEXT: REM DO NEXT SUCCESSION
   IN THE LOOP
```

Many programming books and articles recommend using REM's sparingly, since it is claimed that they slow down a program, but as you'll see in the section on program speed, it's really the way in which REMs are used, rather than the REM's themselves, that is responsible for putting the brakes on otherwise speedy software.

MOVING AHEAD

The Commodore 128 offers many other BASIC commands that we cannot begin to cover here. In fact, there are more than 100 BASIC commands available. We'll touch on many of these in the body of this book. The others are up to you. Never be afraid to experiment. It's the key to learning.

Appendix B

A Complete Relative File Program

This program is an example of a relative (random) file program using a professional-looking bar menu and form locations for entry.

Additional routines from the pages of this book (to search for and sort information) can be easily added. To perform a global search through all fields in the file you would design a routine that would do the following:

1. Read each field in each record, one record at a time.
2. Compare the contents of each field with the search item.
3. Set a FOUND flag and return if the item is found.

For a specific search through a single field you would do the following:

1. Read a single field in each record, one record at a time. This read will naturally be based on the PLACE array (if FIELD indicated the field through which you were searching, the file pointer would be positioned at PLACE-(FIELD)).
2. Compare the contents of each field with the search item.
3. Set a FOUND flag and return if the item is found.

This relative filing system uses routines already covered in detail through this book. The only change is the input routine, which has been modified slightly for this application.

```
1  ::::::::::::::::::::::::::::::::::::
2  :
3  : rem random file example
4  :
5  rem trap 20000     rem        open error
6  gosub 60000: rem variables
7  color 5,2      :rem assumes text grphics
10  scnclr
20  gosub 14000   : rem open
30  gosub 50000   : rem menu
```

```
60  dclose
99  end
2400  :rem formatted entry
2410  :for x=1 to fields
2420  :     char ,c(x),r(x),head$(x),1
2430  :next
2435  :
2440  :return
2500  :rem clear previous fields
2505  :
2510  :for x=1 to fields
2520  :     a$(x)=" "
2530  :next
2535  :
2540  :return
3000  ::::::::::::::::::::::::::::::::::
3005  scnclr
3010  rem                              add a record
3012  again=-frst=-1
3015  :
3020  do until not again
3025  if frst then frst=0
3026     gosub 2400:gosub 2500  :rem show
      fields/clr data
3030     add=-1       :  rem      add flag
3040:
3050:   gosub 4000 : rem      input rtn
3052:
3055  :  if esc then exit
3060  :
3070  :    nr=nr+1      : rem bump# of recs
3080
3090  :  rec=nr
3100  :
3110  :    gosub 12000: rem     write record
3115  :
3120  :    char ,0,23,el$+"add another"
3130  :    input y$
3135  :
3140  :    if  instr("Yy",left$(y$,1))  >0  then
      again=-1:else again=0
3150  loop
3155  esc=0    :rem clear
3160  :
3165  added=0
3170  return
3400  ::::::::::::::::::::::::::::::::::
3410  rem                              change a record
3420  again=-1
3430  :
3440  :    do until not again
3450  rec=0
3460  :    do until rec >0 and rec<=nr
3470  :     scnclr:input"record (#)";rec
3480  :    loop
3485  :    gosub 3500       :  rem   edit rec
3487  :    if esc then exit
3490  :    char ,0,23,el$+"change another"
3491  :    input y$
3492  :    if  instr("Yy",left$(y$,1))  >0  then
      again=-1:else again=0
3494  loop
3495  esc=0   :rem clear
3496  :
3498  return
3500  ::::::::::::::::::::::::::::::::::
3510  rem                                  edit a record
3520  :
3530  :
3540  gosub 13000: rem read rec
3550  :
3560  gosub 2400 : rem disply data
3570  :
3580  gosub 4000 : rem input rtn
3582  :
3585  if esc then 3640
3590  :
3600  gosub 12000: rem     write record
3610  :
3620  :
3630  :
3640  return
3800  ::::::::::::::::::::::::::::::::::
3810  rem                                  display data
3820  :
3825  scnclr
3830  for item=1 to fields
3840  :  char,0,item,head$(item)+"   "+a$
      (item)
3850  next
3860  return
4000  rem : overall input routine :
```

```
4010  :ok$="n"
4020  :do while ok$="n"
4025  :    gosub 4300    :rem clr /display data
4026  :    item=0:esc=0
4030  :    do until esc or item=fields
4031  :         item=item+1
4032  :
4035  :         le=le(item)    :rem length
4036  :         v=r(item)+1:h=c(item)  :rem
        row,col
4040:         char ,c(item),r(item)+1
4050:         bg$=a$(item) :rem background data
4060:         gosub 5000 :rem input routine
4062  :       if in$>" " then a$(item)=in$:else
        char ,h,v,a$(item)+reset$
4065  :         gosub 4500
4070  :    loop
4072  :    if esc then exit
4075  :    gosub 4300: rem display data
4080  :    char ,0,22
4090  :    input "ok?(y/n)";ok$
4100  :    if instr("yY",left$(ok$,1)) then ok$="y"
4110  loop
4115  :
4120  return
4300  : rem   :clear data rtn:
4305  :
4310  :for item=1 to fields
4317  :     dum=len(a$(item))
4320  :     char ,c(item),r(item)+1,a$(item)
4322  :     if add then begin
4324  if add then char ,c(item)+dum,r(item)
        +1,left$(ulin$,le(item)-dum)
4327  if not add then char ,c(item)+dum,
        r(item)+1,left$(space$,le(item)
        -dum)
4328  :     bend
4329  :
4330  :next
4335  :
4340  :return
4350  :       char ,0,23,"item too long—please
        re-enter"
4360  :       char,0,item,el$
4370  :       big=-1       :rem          redo
4380  bend:else big=0
4390  return
4400  :
4500  ::::::::::::::::::::::::::::::::
4510  rem                          check length
4520  :
4530  if len(a$(item)) > le(item) then begin
4540  :       play "sceg"
4550  :       char ,0,23,"item too long—please
        re-enter"
4560  :       char,0,item,el
4570  :       big=-1       :rem          redo
4580  bend:else big=0
4590  return
4599  :
5000  ::::::::::::::::::::::
5020  rem                          input routine
5040  :
5050  tempo 255
5060  gosub 5900              : rem clr buffer
5080  in$=" ":ch$=" ":bg$=" ":in=0:dun=0
5085  :
5090  if le>240 then le=240
5100  :
5120  do until dun
5122  :  char ,h,v,case$+in$+csr$+reset
5125  :  ch$=" ":do while ch$=" "
5130  :  char ,h,v,case$+bg$+reset
5132  :  char ,h,v,case$+in$+csr$+reset
5140  :  get ch$:loop
5160  :  ch=asc(ch$)
5180  :  gosub 5400       : rem    test ch
5190  :  if dun then exit
5200  :
5210  :  if bks then bks=0:gosub 5950
5300  :  if in then in$=in$+ch$:if not add then
         bg$=in$+left$(ulin$,le-ln+(le>ln))
5390  loop
5392  :  char ,h,v,case$+in$+"  "+reset
5395  return
5400  :
5410  dun=0:in=-1:bks=0:esc=0
5415  :
5420  rem                          test character
5430  ln=len(in$): rem cur len
5440  if ch=13 then dun=-1:in=0: gosub 5800:
        rem cr
```

```
5460  if ch=27 then dun=-1:in=0:esc=-1:rem
        esc
5480  if (ch>144 and ch<149) or ch=157 or
        ch=95 or (ch<32) then bks=-1:in
        =0:goto 5530
5500  if ch>96 then ch=ch-128:ch$=chr$(ch):
        goto 5520
5505  if nmbr and (ch>57 or ch<45) then play
        bleep$ :in=0:goto 5530
5520  if ln=>le then play bleep$:in=0
5530  return
5600  ctr=ctr+1
5605  tempo 100
5610  play "o2"+mid$(music$,ctr,1)
5620  if ctr=lmt then ctr=0
5625  tempo 255
5630  return
5700  ::::::::::::::::::::::::::::::::::::
5710  rem                display time
5720  char ,0,24,left$(ti$,2)+":"+mid$(ti$,3,2)
        +":"+right$(ti$,2)
5730  return
5800  ::::::::::::::::::::::::::::::::::::
5805  rem              clear to end of field
5810  : char ,h,v,case$+in$+left$(space$,
        le-ln)+reset
5820  :
5830  return
5900  :
5905  rem              clear buffer
5910  forzz=1 to 8:get ch$:next
5920  return
5950  :
5960  :rem             backspace rtn
5970  if ln=0 then tempo 255: play"s v1o5 c v2o6
        c v3o4c":goto 5995
5980  in$=left$(in$,ln-1)
5995  return
12000 ::::::::::::::::::::::::::::::::::::
12010 rem              write to file
12015 rcrd=rec+1
12020 :
12030 record#1,rcrd
12040 for item=1 to fields
12045 :   record#1,rcrd,place(item)
12048 :   print#1,a$(item)
12050 :
12080 next
12090 if added then gosub 12300
12299 return
12300 ::::::::::::::::::::::::::::::::::::
12305 :rem          write # of records
12310 :
12320 record#1,1:record#1,1
12330 :
12340 print#1,nr
12350 :
12360 return
13000 ::::::::::::::::::::::::::::::::::::
13010 rem            read from file
13015 rcrd=rec+1
13020 :
13030 record#1,rcrd
13040 for item=1 to fields
13045 :   record#1,rcrd,place(item)
13048 :   input#1,a$(item)
13050 :
13080 next
13299 return
14000 ::::::::::::::::::::::::::::::::::::
14010 rem            normal open
14020 :
14030 dopen#1,(file$),l(lr)
14032 :
14035 rcrd=256/lr+1: rem next sector
14037 :
14040 record#1,rcrd:record #1,rcrd
14050 :
14060 if ds=50 then gosub 14100:scnclr : rem
        create new file
14065 :
14070 record#1,1:record#1,1:rem # of recs
14075 :
14080 input#1,nr
14099 return
14100 ::::::::::::::::::::::::::::::::::::
14110 rem            create if not exist
14120 :
14125 scnclr:char ,1,12,"creating the "+file$+"
        file"
14130 record#1,100:record#1,100:  rem 100
        records
```

```
14140  print#1,chr$(255): rem null record
14145  record#1,1:record#1,1:rem # of recs
14148  print#1,0: rem no rcrds yet
14150  dclose#1
14160  dopen#1,(file$),l(lr)
14199  return
50000  : rem menu using on . . . gosub
50001  : read ndex$ :rem index to keys
50002  : do while select$(ctr)< >"###"
50003  :     ctr=ctr+1
50005  :     read col(ctr),row(ctr),select$(ctr)
50006  : loop
50007  : no=ctr-1 :rem nbr of items
50009  : do until esc
50010  :  cnclr:gosub 52000     :rem display full
50012  :  choice=1:last=no
50015  :  gosub 54000 :rem display highlgt
50017  :  retrn=0:esc=0
50020  :  do until retrn or esc
50030  :     gosub 53000 :rem get key
50040  :     gosub 54000 :rem display partial
50050  :  loop
50095  char ,0,22
50100  :    on choice gosub 3000,3400
50132  :  if choice=3 then esc=1
50133  ::::::::::::::::::::::::::::::::::::::
50135  loop
50140  print:print"bye!          "
50150  return   :rem return to close file
50160  :
52000  : rem display all options
52010  :
52020  : for ctr=1 to no
52030  char ,col(ctr),row(ctr),select$(ctr),0
52040  next
52050  return
52060  :
53000  : rem    keypress
53005  : last=choice
53010  : getkey  a$
53012  :
53015  if instr(ndex$,a$) then choice
           =instr(ndex$,a$)
53020  : if a$=up$ then choice=choice-1
53030  : if a$=dwn$ then choice=choice+1
53035  if a$=esc$ then esc=1:choice=no
53040  : if a$=retrn$ then retrn=1
53045  : if choice<1 then choice=no
53047  : if choice>no then choice=1
53048  :
53050  : return
53060  :
54000  :rem display bars
54010  :
54020  char ,col(last),row(last),select$(last),0
54030  char ,col(choice),row(choice), select$
         (choice),1
54035  char ,9,22,"esc to end"
54040  return
54050  :
55000  :rem menu data
55015  data "ace" :rem first ltr index
55020  data 5,10,"    add a record       "
55030  data 5,11,"    change a record    "
55060  data 5,12,"    end program        "
55200  data 0,0,"###"    : rem   end mark
60000  ::::::::::::::::::::::::::::::::::::::
60010  rem                  opening assignments
60020  head$(1)="first"
60030  head$(2)="last"
60040  head$(3)="home phone"
60050  head$(4)="office phone"
60060  :
60070  place(1)=1 : le(1)=10
60080  place(2)=12 : le(2)=20
60090  place(3)=33 : le(3)=13
60094  bleep$="s v1o5 g v2o6 g v304 g"
60095  space$=" "
60096  space$=space$+space$+space$
60097  csr$=" "
60098  case$=chr$(142)
60099  reset$=chr$(27)+"c"
60100  place(4)=47 : le(4)=13
60110  :
60120  lr=64          : rem length of rec
60125  fields=4
60140  file$="test"
60145  quote$=chr$(34)
60147  ulin$=chr$(164):forx=1 to 5:ulin$=
         ulin$+ulin$:next :rem create
         underline
60148  el$=chr$(27)+"q"
```

```
60150  :
60152  scnclr
60155  dwn$=chr$(17)
60160  up$=chr$(145)
60165  retrn$=chr$(13)
60170  esc$=chr$(27)
60171  :
60172  c(1)=0:r(1)=1
60174  c(2)=15:r(2)=1
60176  c(3)=0:r(3)=4
60178  c(4)=15:r(4)=4
60180  dim a$(4)
60182  open 2,0: rem open keyboard, allowing in-
           put without ? mark
60190  return
60199  return
61000  return
```

Index

Other Bestsellers From TAB

☐ **1001 THINGS TO DO WITH YOUR COMMODORE® 128—Sawusch and Prochnow**

Expert software developer and computer writer Dave Prochnow has expanded and improved on the exceptional work done by computer pioneer Mark Sawusch to produce a collection of applications and programs that explores the outer limits of C-128 capabilities. Packed with ingenious and innovative ideas for putting your computer to work—including lots of ways to save time and money, even ways to use your computer to make money—it's a book that's sure to inspire you to come up with still more ideas of your own. 208 pp., 167 illus. (7″ × 10″).

Paper $12.95 **Hard $18.95**
Book No. 2756

☐ **COMMODORE 64™/128™ GRAPHICS AND SOUND PROGRAMMING—2nd Edition—Krute**

Here's all the hands-on, learn-by-doing information you'll need to start taking full advantage of your Commodore's exceptional graphics powers—sprite, character, and bit-mapped graphics. Plus, you'll find out how to utilize all of your machine's advanced three-voice music synthesizer chip. Best of all, you'll find a whole collection of ready-to-run programs to demonstrate how each concept works on both the C-64 and the C-128! 272 pp., 157 illus. (7″ × 10″).

Paper $14.95 **Hard $22.95**
Book No. 2640

☐ **COMMODORE 64™ EXPANSION GUIDE—Phillips**

Far more than just a product listing or a rehash of manufacturer's sales brochures, these are the best of the hundreds of hardware accessories currently on the market . . . each one chosen for value and performance after exhaustive testing and examination. You'll find in-depth background on each type of device—printers, disk drives, modems, monitors, and photographic details. 288 pp., 31 illus. 7″ × 10″.

Paper $16.95 **Hard $22.95**
Book No. 1961

☐ **THE COMPUTER FURNITURE PLAN AND PROJECT BOOK—Wiley**

Now, even a novice can build good looking, functional, and low-cost computer furniture that's custom-designed for your own special needs—tables, stands, desks, modular or built-in units, even a posture supporting kneeling chair! Craftsman Jack Wiley provides all the step-by-step guidance, detailed project plans, show-how illustrations, and practical customizing device. 288 pp., 385 illus. 7″ × 10″.

Paper $15.95 **Hard $23.95**
Book No. 1949

☐ **THE ILLUSTRATED DICTIONARY OF MICROCOMPUTERS—2nd Edition—Hordeski**

Little more than a decade after the introduction of the first microprocessors, microcomputers have made a major impact on every area of today's business, industry, and personal lifestyles. The result: a whole new language of terms and concepts reflecting this rapidly developing technology . . . and a vital need for current, accurate explanations of what these terms and concepts mean. Michael Hordeski has provided just that in this completely revised and greatly expanded new second edition of *The Illustrated Dictionary of Microcomputers*! 368 pp., 357 illus. (7″ × 10″).

Paper $14.95 **Hard $24.95**
Book No. 2688

☐ **PRACTICAL INTERFACING PROJECTS WITH THE COMMODORE™ COMPUTERS**

Hands-on techniques for transforming your C-64, C-16, Plus/4, VIC-20 or the new C-128 into an accurate controller for science, engineering, or home and hobby electronics applications.Includes over 80 different software programs—This is a sourcebook that will have you using your Commodore computer in some truly exciting new ways. 256 pp., 256 illus. 7″ × 10″.

Paper $16.95 **Hard $24.95**
Book No. 1983

☐ **101 PROGRAMMING SURPRISES AND TRICKS FOR YOUR COMMODORE 64 COMPUTER**

This exciting new collection of games, novelties, and programming marvels is fresh, literate, and packed with all kinds of downright amazing ways to have fun with your C-64. And unlike other programming books, it makes no attempt to instruct you—instead, the object is to entertain and be entertained. 224 pp., 12 illus. 7″ × 10″.

Paper $11.95 **Hard $18.95**
Book No. 1951

☐ **COMMODORE 64™ ADVANCED GAME DESIGN—Schwenk**

Professional game designers George and Nancy Schwenk reveal their winning formula for creating stimulating, professional-quality microcomputer games for family fun and even profit! Using three fully-developed C-64 games to illustrate game design, this innovative tutorial provides an informative and practical look at the conceptual and implementation techniques involved. 144 pp., 14 illus. 7″ × 10″.

Paper $10.95 **Hard $15.95**
Book No. 1923

*Prices subject to change without notice.

Look for these and other TAB books at your local bookstore.

TAB BOOKS Inc.
P.O. Box 40
Blue Ridge Summit, PA 17214

Send for FREE TAB catalog describing over 1200 current titles in print.

Commodore 128™ BASIC: Programming Techniques

Martin Hardee

The one source on C-128 BASIC that you'll turn to again and again for information and inspiration!

You don't have to spend a fortune on a computer in order to get the programming power you need! Now, with the help of this user-friendly guide, you can get unbelievable programming productivity and versatility using BASIC on your C-128! You'll learn disk commands that will allow you to store and retrieve data, telephone communications techniques that will broaden your horizons, and sound and graphic commands that will make your programs more enjoyable and friendly.

Let an expert on the C-128 show you a host of commands and tricks that make programming in BASIC on the C-128 easier and faster than you ever thought possible. A part of the team that developed the "manual on disk," now included as part of the Commodore 128 package, Hardee takes a candid look at the C-128 from the viewpoint of a new *user*, not from the standpoint of an engineer or programmer. Writing in a direct, jargon-free style, he reviews the basics and moves you on your way to more advanced applications that make your programs work harder for you. Even if you're familiar with BASIC on the C-64, you'll still want to learn all the improved aspects of the language that are within your reach using the C-128.

Starting with an introduction to general disk commands, you'll learn everything you need to know to harness the new, "plain English" disk commands, make programs more colorful without spending a lot of time, and make programs both efficient and easy to read. You'll find out how to use both character modes on the same screen, how to use simple shapes for impressive results, and how to mix text and graphics to give your programs a more attractive and professional appearance. Plus you'll get complete instructions for input formatting. In a class by itself, this invaluable guide features over 50 practical programs for a wide range of applications!

Martin Hardee is a professional technical writer and programmer who has worked on projects for Tandy (Radio Shack), Sanyo, Kaypro, and many other leading computer firms.

TAB TAB BOOKS Inc.

Blue Ridge Summit, Pa. 17214

FPT > $12.95

ISBN 0-8306-2732-4

PRICES HIGHER IN CANADA

1260-0886