SERIOUS PROGRAMMING FOR THE COMMODORE 64

HENRY SIMPSON

SERIOUS PROGRAMMING FOR THE COMMODORE 64

HENRY SIMPSON

TAB TAB BOOKS Inc.
BLUE RIDGE SUMMIT, PA. 17214

FIRST EDITION

FIRST PRINTING

Printed in the United States of America

Library of Congress Cataloging in Publication Data

Simpson, Henry.
Serious programming for the Commodore 64.

Includes index.
1. Commodore 64 (Computer)—Programming. 2. Basic (Computer program language) I. Title.
QA76.8.C64S54 1984 001.64'2 84-16450
ISBN 0-8306-0821-4
ISBN 0-8306-1821-X (pbk.)

Contents

Preface

The Commodore 64 is a marvelous machine. It is both powerful and inexpensive—definitely a winning combination. Software developers and hobbyists alike have discovered that it is a true computer and not a toy. While it is a terrific game machine, it is also good for running more "serious" application programs. Many folks find it more useful to balance their budget or scan their spreadsheet than to zap descending aliens.

This book focuses on the development of such "serious" programs. I hope that the subject does not sound dull. Serious, after all, does not mean *humorless* as much as *to be taken seriously*. In the realm of software development, serious translates to mean something like *professional*. This book, then, provides guidelines and specific techniques for developing serious programs that reflect some degree of professionalism.

What makes a serious program? In the recipe include user-friendliness—that is, making the program safe for users, and fairly easy to learn and use. Another ingredient is designing the program so that it is friendly to programmers—understandable, well documented, and maintainable. The serious program makes full use of the resources available to it—the BASIC language, DOS (disk operating system), assembly-language routines—and whatever other tricks the programmer can invent, discover, or borrow from others. Finally, such a program is usually developed systematically, not while the programmer is expressing momentary inspirations when seated before a video display.

The most important thing about a serious program is that its author has serious ambitions—a desire to perform a useful task in a professional manner. When you have this ambition, you discover that there is more to creating such a program than wishing. It takes specific knowledge and specific skills. Some programmers pick up what is needed through experience and training, but sad to say, some programmers never learn the tricks at all.

This book is my attempt to share what I have learned about program development and about the Commodore 64. It is full of general guidelines,

specific techniques, subroutines, program fragments, and everything else that I could contrive to get my message across to you. I have assembled this information from innumerable books and magazine articles, the mental recesses of Commodore experts, and from several years of designing, developing, and evaluating software for a variety of microcomputers. I think that some of these "learning experiences" may be helpful to others, and herewith I share them with you.

Acknowledgments

Special thanks are due to Dave Wiesner for his generous help in reviewing drafts of portions of this book, and for his creative work in writing the assembly-language subroutines Chapter 4 and the program chaining technique in Chapter 7.

My appreciation also goes to Barbara Gates, who not only did the word processing in this book, but also submitted to the indignity of frequent wholesale rewrites and totally unreasonable deadlines.

Introduction

This book was written for people who want to get more from their Commodore 64 computer. It was written for programmers by a programmer. Although it deals with the Commodore 64 (C-64) computer, and all of its examples are for the C-64, most of what it covers goes well beyond the C-64 and will be useful to programmers who develop programs for any computer system.

This book has four themes. The first theme is that programs that people such as yourself design should be "friendly" to their users. To write a friendly program, you must understand your users and their needs. You must then consciously take these needs into account as your design and develop your program. This book shows you how to do these things.

The second theme is that the programs you design should be friendly to programmers. By this I mean that your program should be logically organized, readable, well documented, and maintainable. To write a program that allows this, you must make a special effort to do certain things. The payoff is that your program will not be a mystery to you 6 months after you wrote it, and that it will be understandable to others. These things are important if you want to fix a bug or modify the program.

The third theme is that it is important for you to be in control of your C-64. You should be able to make it do what you want it to do. If, for example, you want to move a cursor to a particular part of the screen, read a text file, or initialize a disk within a program, you should be able to do these things. There are two parts to this. First, the computer's software—DOS (disk operating system) and BASIC language—must permit you to do these things. Second, you must know what incantations to invoke, lines of code to type in, and buttons to push to make the magic happen. The C-64 software leaves something to be desired. The 2.0 level BASIC lacks some important commands found in most business BASICs, and the DOS uses a confusing syntax and has a few gremlins in it. The majority of these shortcomings can be overcome by using the programming techniques covered in this book. (To be fair, the C-64 has many features not found in other microcomputers—such as superb

color graphic and sound generation.) To put it another way, I offer you a bag of tricks to use. Before you can use them, you must learn them, and this may take a little effort. Not too much, I hope—but if you apply the effort, you will learn the tricks (*programming techniques* is a more dignified term, but tricks is what they are) and be able to make your C-64 do things that you did not know it could do.

The fourth theme is that programs should be developed systematically and according to a strategy. Software engineering is not a very exciting subject for most people, and I will try not to bore you too much with it in this book, but there are some important ideas here that bear emphasis. Among these ideas is the "top-down" programming technique, developing a program as a set of modules, or building blocks, and systematic testing and evaluation. While most of these ideas are abstract, once you understand them they can become very real to you and have an important effect on the way you design and develop a program. Programs that are properly engineered are better—simpler, easier to understand, and less likely to have bugs.

As you can see, of these four themes, only the third deals specifically with the C-64 computer. The other three are applicable to any computer system. This book deals, in part, with C-64 programming, human factors, software engineering, and system and user documentation. There should be something here for everyone.

For whom was this book written? The short answer is that it was written for people who want to develop better programs for the C-64. A more complete answer is that it was written for people who already possess some programming skill on the C-64, who wish to develop serious application programs for it, and who would like to refine their skills as program developers. The book assumes a familiarity with the C-64 and with its BASIC language and DOS. I do not explain such things as how to connect your disk drive to your computer or what a FOR-NEXT loop is. I assume that you already know the elementary facts about your computer. This is not really a book for the beginning programmer. If you are one, then you would be well advised to set this book aside and improve your programming skills before you pick it up again. You do not have to be a programming expert to use this book, but you must know the fundamentals. If you are not quite sure whether you are ready, then forge ahead with this book anyway. I have made a strong effort to explain everything fully and to illustrate points with examples to aid your understanding.

Since I have sketched in general terms what this book covers, it is probably a good idea to tell you what it leaves out. First, I do not have much to say about color, at least not in the sense that it is used in game or entertainment programs. I talk about the use of color for presenting information, and good and bad color combinations for information display. In other words, I focus on what might be called the "practical" factors surrounding color rather than on color for its own sake.

I do not say much about graphics, either. This is a complex subject and I would rather avoid it here, quite frankly. Books have been written on the subject, and it would not be difficult to write one on it for the C-64. I will leave this project to others.

Likewise, this book says nothing about sound. The C-64 has a marvelous sound-producing capability which can be exercised in games, entertainment programs, or the development of music. Alas, the C-64's talent in this area has little practical utility in most of the programs of the type covered in this book.

Mainly, I want to show you how to write programs that will make the C-64 at last into a respectable computer, rather than something many otherwise well-informed programmers regard as a "game machine" or a "toy." The C-64 can do useful things, as many people are discovering. Much serious software is now being published for the C-64—word processors, data base managers, financial analysis programs—and more is being published all the time. The amount is not as great as for the computer named after a fruit, or the one produced by the big company whose name consists of three initials. Nonetheless, there is no reason for the C-64 programmer to accept the stereotype and assume that every program developed for their computer must employ color graphics, make deafning sounds, or require the user to perform some

act of mayhem with a joystick. The C-64 is a real computer and can do all the stuff the others can—if you know what strings to pull to activate its various bells and whistles.

On top of its potential, the C-64 is inexpensive, gaining wide acceptance, and finding its way into an enormous number of homes and small businesses. It and similar machines—the new models introduced by Commodore as well as machines introduced by other manufacturers—deserve software of the same quality as that used in larger, more expensive machines.

My general approach to presenting information in this book is to support each principle with a practical example that you can type into your own computer. Most of the chapters in this book contain several segments of BASIC code—code fragments, subroutines, or short programs—which you can and should try out for yourself. Not only will you learn more this way, but the learning will be more interesting and fun. In addition, if you save the subroutines that appear in each chapter, when you finish the book you will have a subroutine library that is very useful for developing programs of your own.

In general, I have followed the "less is more" philosophy in writing this book. It may not always seem this way, but I have attempted to keep things as simple and straightforward as possible. From my own attempts to learn how to program, I have discovered that most problems were in deciding which information would be useful and which I could safely ignore. Unfortunately, I could only make this judgment after I had learned (or attempted to learn) everything. Then I would throw away about 90 percent of all that stuff I had struggled so hard to master.

The present book does not work that way. Rather than offer you a lot of choices about how to do this or that—center a character string on the display, create a data file, design a menu—I give you one simple, straightforward way that works. Hopefully, this method will make things easier because you will have less to wade through and discard.

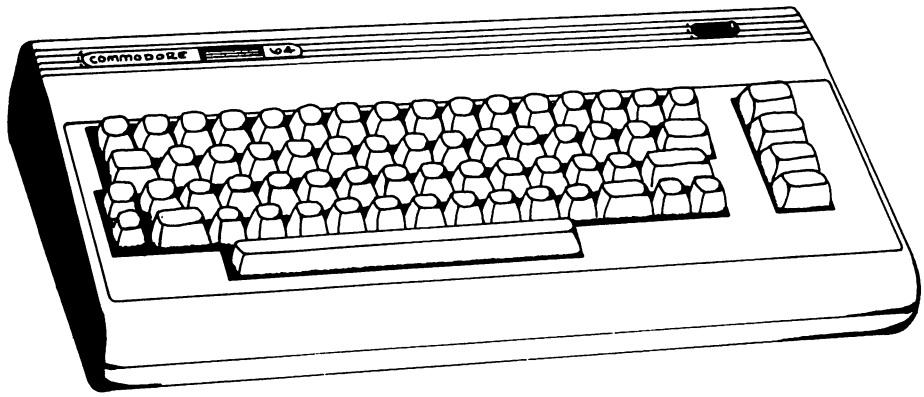
This book is organized in 10 chapters and 2 appendices. Chapters 1 through 3 are introductory and cover subjects you should master before you sit down and start coding a program. Chapter 1 discusses techniques for making your program friendly to users and to other programmers, describes some of the limitations of the C-64 and areas in which it needs enhancement, and sketches a strategy for program development. Chapter 2 offers advice and suggestions on the materials you should acquire before attempting a serious programming project—hardware, software, publications. Chapter 3 offers programming tips and discusses system documentation, and introduces Fred, a world traveling novice programmer in search of certain ultimate (more or less) truths.

Chapters 4 through 9 cover various programming subjects: 4—Output and Screen Design; 5—Data Entry, Error-Testing, and Validation; 6—Program Control; 7—Program Chaining; 8—File Handling; and 9—Using Your Printer. Each of these chapters discusses how to get control of a different aspect of your C-64. Each also offers advice on how to do this in such a way that your program will be friendly to users.

Chapter 10 tells what your program user's guide should contain and how to develop help screens for use within your program.

I suggest you start with Chapter 1 and read straight through to the end, trying out the things I show in each chapter. Later on, you may want to go back to Chapters 4 through 9, which contain the basic reference information in the book.

Chapter 1



User-Oriented Program Design

This chapter introduces the four themes of the book: making your program friendly to users, making your program friendly to programmers, getting control of your C-64, and developing your program according to a strategy. This chapter tells what the themes mean and why they are important, and attempts to raise your consciousness about them. The four themes flow through the rest of the book in various ways.

How important are the ideas underlying these themes? This depends on your programming goals.

If you write programs strictly for your own use, then you probably do not care how "friendly" you make them to yourself in your role as program user or programmer. You probably do care about what I call "getting control of your computer," and improving your strategy for program development.

On the other hand, if you work with other programmers—for example, in a user's group, or on program-development projects—then you probably care very much about making your programs friendly to both users and programmers. You may also feel that you have less to learn about programming or program development.

Whatever your programming goals, this book offers techniques that will make you a better programmer, and this chapter lays the groundwork for the rest of the book. The chapter defines and describes each of the themes and explains its importance. The themes are discussed in the order just listed.

If this does not sound exciting, take heart. In this chapter you will discover, among other things, the basic formula for making programs user-friendly, a catalog of programmer's sins, and a discussion of the shortcomings of the C-64.

MAKING YOUR PROGRAM FRIENDLY TO USERS

"User-friendliness" is something we hear much about these days, although no one has yet given a satisfactory definition of what it is. It is generally regarded as a good thing, like the flag or motherhood, even by those who have only the vaguest notions about it. This makes a handy combina-

tion for advertisers: What this product is is not very clear, but it is good.

It rings like a political slogan. Picture a brightly lighted auditorium, banners waving, and a politician (this one looks like a computer nerd), standing on a platform before a large audience, proclaiming something such as this:

> "In the past, this nation has paid too little attention to the program user! We have not been friendly to users! We have promised, but not delivered. In our administration, we will see that users get what is coming to them! We will appoint a blue-ribbon panel to investigate past abuses of users, and make recommendations. We will then act, in a fair and deliberate manner. Our goal is to eliminate user abuse within this generation." And so on . . .

A bit fanciful, perhaps, but no less so than the typical ad for a user-friendly program. You know. You have seen it. The ad proclaims that a program offers the ultimate in user-friendliness because it uses a mouse, can be mastered in 5 minutes, and can be used by an 8-year-old child.

What, if anything, do such claims have to do with reality? Consider. Since the idea of user friendliness has never been adequately defined, people have diverse, fragmented, and sometimes strange ideas about what it means.

For example, one programmer has the idea that friendliness means protecting the user against all possible data-entry errors. To ensure that errors are prevented, each time the user types something in and presses the Return key, the programmer has the computer come back with the following on-screen prompt:

ARE YOU SURE? (Y/N): _

When the user types in "Y," this prompt appears:

ARE YOU REALLY,
REALLY SURE? (Y/N): _

This prompt is helpful if the user has made a mistake, but becomes maddening after the user masters the program.

Another programmer's idea of user-friendliness is menus. Menus, the programmer believes, are good things. There is something magical about them that makes programs friendly. There is an element of truth in this, as in the idea that users should be protected from data-entry errors, but both of these ideas are oversimplifications of what user-friendliness is. The menuphile attempts to use a menu everywhere, even in places where it is unnecessary and undesirable. Menus can be helpful for certain things, but they also can be slow, especially for experienced users. Thus, a fixation on them, as on any single thing in life, is unhealthy.

A third programmer is into color. This programmer uses color on every display. The resulting programs are a visual delight, although sometimes the color combinations used produce apparent shows and afterimages, or the contrasts make it difficult to read things. Well, color can be useful, but using it properly is actually quite tricky and requires some knowledge of human color perception. Another programmer is into graphics, another into icons, another into windowing, and so on.

I could go on further along this line, but rather than dwelling on misconceptions and errors, it is more productive to present a more accurate picture of what user-friendliness truly means.

Start with the User

In designing anything that people will use, it is a good idea to find out as much about the users as possible. That way you can tailor your design to fit their needs.

You already do this, often unconsciously, in many ways. When you talk to a small child, you use simple words and short sentences, and pause frequently to make sure that the child understands.

When you barbecue steaks, you ask your guests how they want theirs—well-done, medium, or rare—to make sure that they get what they like. (If your cooking skills are similar to mine, your steaks are not always "user-friendly," but at least you attempt to make them that way.)

When you write a scholarly paper for presentation in a journal, you use the particular disipline's technical vocabulary and leave many things unstated, since your audience should already know them. Your paper will probably be incomprehensi-

ble to the uninitiated, but your intended audience should have no difficulty with it.

When you write a letter to your rich and straight-laced Uncle Oscar, you avoid strong language, complimentary references to liberals, or saying anything nice about the Russians, since you know from experience that he is sensitive about these subjects and may cut you out of his will.

These are examples of ways in which you take your audience into account in your daily life. Most of this is unconcious, as I said, but some of it is very deliberate. For example, in writing to Uncle Oscar, you may go through a sort of mental checklist just to make sure that you have censored your letter properly. Talking to the small child is done more casually, and you probably do not think about what you are doing very much. In each of these cases, however, consciously or not, you recognize that your audience has a certain need and that you must tailor the information you provide to meet that need. This is the essence of user-friendliness.

It is not as easy to write a friendly program as it is to write a friendly letter. In fact, the whole process may seem a bit foreign.

Where do you start? With the user. Pin down your user as accurately as you can. Ask yourself questions such as:

- How much computer sophistication will the user have?
- How much will the user know about how computer programs operate?
- How much will the user know about the theory behind your program? (For example, if you are writing an accounting program, how much accounting sophistication will the user have?)
- How intelligent will the user be?
- Will the user have any handicaps or impairments (such as color-blindness) that influence the way the program can be used?

This list of questions is by no means complete, but it will give you an idea of the kinds of questions you must ask before you start program design. Once you know who your users are, you can determine their needs and design your program in such a way that these needs will be met.

For example, if your users will be children or adults who lack computer sophistication, then you must provide a good deal of on-screen prompting in your program to ensure that things do not go amiss. With sophisticated users, you can do less hand-holding, and worry more about making your program fast and efficient. If your users will have handicaps or impairments, then you must design your program so that these factors do not interfere with their use of the program. For example, if your users will mainly be males, you should not design your program's displays so that they require color vision. Approximately 10 percent of the male population is color blind.

This is what I mean by taking the user into account. It is not difficult, but to many programmers it is new. Programmers often take the user for granted. They design a program for themselves and assume that everyone has the mental equipment and skills that they have. They show about as much sensitivity to the user as attempting to explain macroeconomic theory to a small child, or writing a letter to rich Uncle Oscar about your respect for the accomplishments of Leon Trotsky. A programmer must have better sense.

The User Learning Curve

An interesting thing happens as people gain experience and skill at doing something. With practice and a good teacher, they do things more rapidly and accurately as time goes on. Actions which once required careful thought and deliberate movements become automatic and rapid. Decisions are made quickly and with what seems to be little conscious thought.

You have seen it happen to yourself and to others. There is no mystery about it. The way in which performance improves with experience is sometimes referred to as the *user learning curve.* This term derives from research in experimental psychology, where some measure of performance (such as accuracy or speed) is plotted against experience, and a characteristic curve is generated (Fig. 1-1). The form of this curve is similar for many

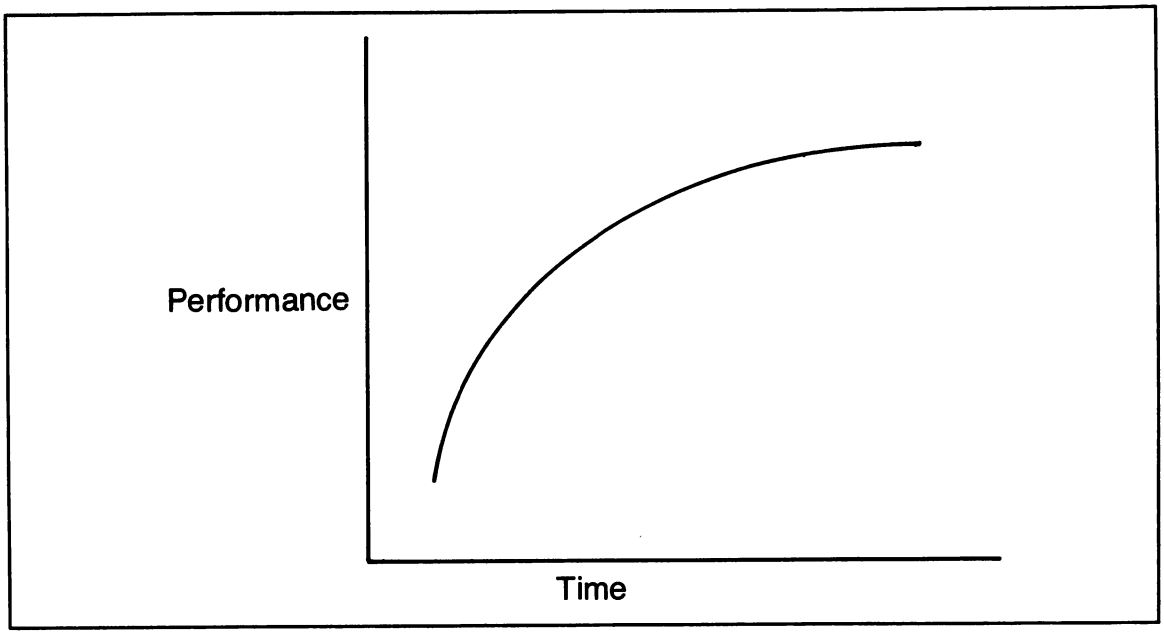


Fig. 1-1. User learning curve. As users gain experience, their performance typically improves in the manner shown here. Performance improvement is the greatest at the beginning.

different kinds of tasks, and so you can sort of depend on it applying in most human learning. The shape of this curve has a very simple interpretation: As people gain skill, they become faster and make fewer errors.

The change in performance is often dramatic. For example, if you design a simple program for computing, say, the monthly payment to fully amortize a loan of a particular principal amount, interest rate, and term, the user may be slow at using it at first, but several times faster once the program is mastered. The more complex the program, the longer it takes the user to master it, and the more drawn out the learning curve will be. Some programs are so complex that the user never attempts to master them fully, but uses only the portions of interest to him. In such cases, the learning curve never actually flattens.

Generally, when the user masters a program, he uses it differently than when he first starts. In the early stages, there is usually a lot of trial and error, caution, and anxiety. Later, the caution and anxiety disappear and the user looks for speed. Program features which aid the novice user, such as obligatory help screens, become obstacles that slow down the program. Some users may in fact decide at this point that the program is not as good as it seemed earlier, forgetting how the help screens brought them to their current level of skill! In a way, you cannot really design a program for a single user, even if that user is one person, because the user will change with increasing skill.

This fact leaves you in a quandary. If you design the program so that it is easy to learn—with a lot of prompting, help screens, and so forth—the user will at some point outgrow it and find parts of it tedious. If you take the opposite approach, and provide very little on-screen help, the program will be more difficult to learn. In short, a trade-off is involved between ease of learning and ease of use.

In some cases, it is best to make the trade-off in favor of ease of learning. In other cases it is best to make it in favor of ease of use. The best way, of course, is to make it possible for the program to be both easy to learn and easy to use—that is, to change depending on the user skill level. One way of doing this is to make help screens available, but optional. Another is to let the user select a program

either with a menu (when inexperienced) or by typing in a program-calling code (when experienced). Besides these, there are other techniques for taking the learning curve into account which will be discussed later in the book. The main thing, for now, is for you to be aware that users learn, and change, and that you need to consider their changing needs in designing your program.

What to Expect from Program Users

Every programmer has anecdotes about the silly, stupid, or in some cases disastrous, things that have happened while using a computer program. For example, a program user calls the programmer and says that he or she followed the procedure in the program user's guide exactly, step by step, but could not get the program started. After a great deal of discussion, it dawns on the programmer that the user never pressed the Return key after typing in entries. "Why not?" the programmer asks. "The user's guide doesn't say anything about that," responds the frustrated user.

I once received a call from a user who was concerned that it was taking too long to compress a file, a procedure that should have taken a few minutes. I asked the user how long the program had been attempting to perform the compression. "About 2 days," came the reply. The user then asked if this was too long.

Have you ever seen someone get mayonnaise on a diskette? Fold one in half? Insert it into a disk drive sideways? All of these things, and many more besides, have occurred. I have no doubt that readers of this book can tell stranger tales than these—all of which brings me to the point: Expect nothing of users.

It is not their fault, nor is it up to them to know everything about computers or about your program. It is your responsibility as a programmer to minimize what they need to know and to tell them everything that they must know. Above all, protect users from the consequences of their own ignorance.

Once you accept these ideas—and most experienced programmers eventually do—then it changes the way you approach program design. You realize how important it is to error-test user data entries. You worry about what will happen to your program if the user presses the wrong key at the wrong time—for example, pressing the RUN/STOP or RESTORE key on your C-64 while using your program. You then take action to prevent the anticipated disaster before it occurs.

Respect your users and protect them. Assume that they will do everything incorrectly. If you want to be safe, assume that they are out to get you, and will find ways to make your program crash or misbehave. It does not hurt to be a little paranoid when designing your program. Afterward, when people use it, you will discover whether you were paranoid enough.

Other Ways to Take Users into Account

User interaction with a program is a sort of conversation. The user makes inputs, usually through the keyboard. These inputs may be data, or they may be commands that control what the computer does. The computer carries on its side of the conversation through its displays. In other words, the user talks to the computer by typing in a command, the computer processes the command, and then it replies by displaying something on its screen or printer. The basic idea of a "user-computer conversation," is illustrated in Fig. 1-2.

The first part of the conversation is user input. Input may be made through the keyboard or any other input device that is connected to the computer—joystick, lightpen, trackball, voice input, or whatever. Input is what occurs when the user enters information into the computer. The user, as already noted, is imperfect and will make mistakes. Thus, the program must filter the user's inputs and only accept those that are legal. This is done by error-testing. You must therefore design various input filters, or error tests, to sanitize the user's inputs.

The second part of the conversation is output—the information that the computer presents to the user. In most cases, this output appears on the computer's video display. Output can, of course, be presented in various other ways, such as via a speaker or by controlling a servomechanism.

Whatever the type of display, you must ensure that it is clear and understandable. You must design displays that the user can understand and use effectively. There are good ways and bad ways to design displays, and you must know the difference. Fortunately, there are many design guidelines, especially for screen and hard-copy displays.

User-computer conversation occurs during program control. Program control is the way in which the user interacts with the program to make it do something. For example, one common method of control is to use program selection menus. In a menu-driven program, the user selects one of several displayed menu options by using the keyboard or a pointing device such as a lightpen, and the computer then executes the program that the user selected (Fig. 1-3). Menu selection is but one of many possible methods of program control. It has advantages and disadvantages, just as do all methods of control. As a programmer, you need to know about the different control methods and their advantages and disadvantages, both from the user's point of view and in terms of program efficiency. Many people automatically think that all microcomputer programs should use menus. Not so. It is a big world out there, and menus are but one of the possibilities open to you.

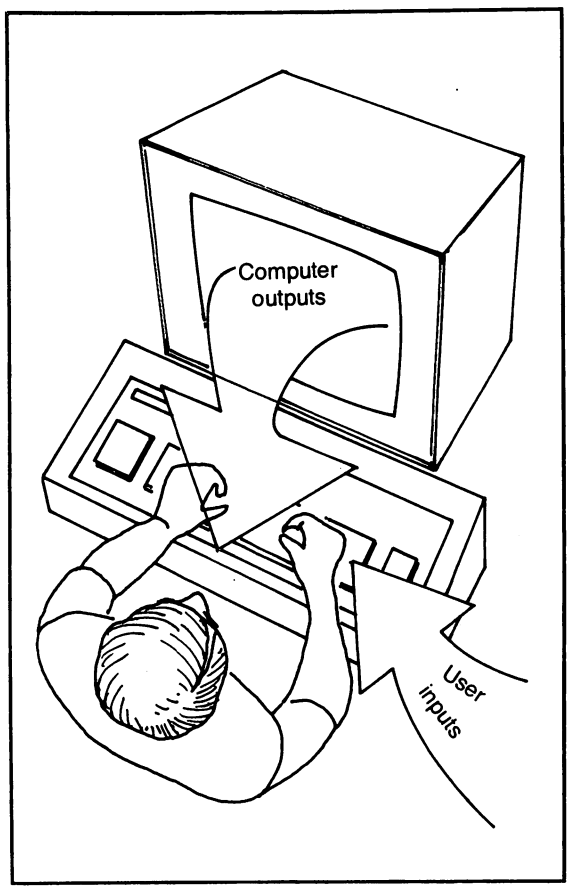


Fig. 1-2. User-computer conversation. The interaction between a user and a computer amounts to a conversation, with the user carryaing on one side of the conversation by making inputs, usually through the keyboard, and the computer carryingon the other side with its outputs, usually through a video display.

In these three areas—input, output, program control—there are friendly and unfriendly ways to design your program. During design, it is difficult to separate the writing of code from the business of making your program friendly. You need to learn the rules of friendliness and then follow them as you code. Subsequent chapters of this book will give you the rules; key chapters are 4, 5, and 6.

Debugging Your Program

No program with bugs in it is friendly. If the program crashes under certain conditions, loses data, presents messy displays, or does other things that it should not do, then it is not yet ready for users. The program that goes to a user should be bug free and impeccable.

It is popular these days to talk about two basic styles of program development. In the quick style, a program is sort of slapped together and then debugged afterward. In the deliberate style, a good deal of advanced planning goes on in an attempt to structure the program so that bugs in the final code are minimized.

The second style is the best because it prevents many errors from finding their way into a program in the first place. There is more front-end work because of the planning that is required, but it actually saves work in the end. A strategy for program development is described later in this chapter. Subsequent chapters in the book—particularly 4 through 9—correspond to steps in the strategy and fill in the details.

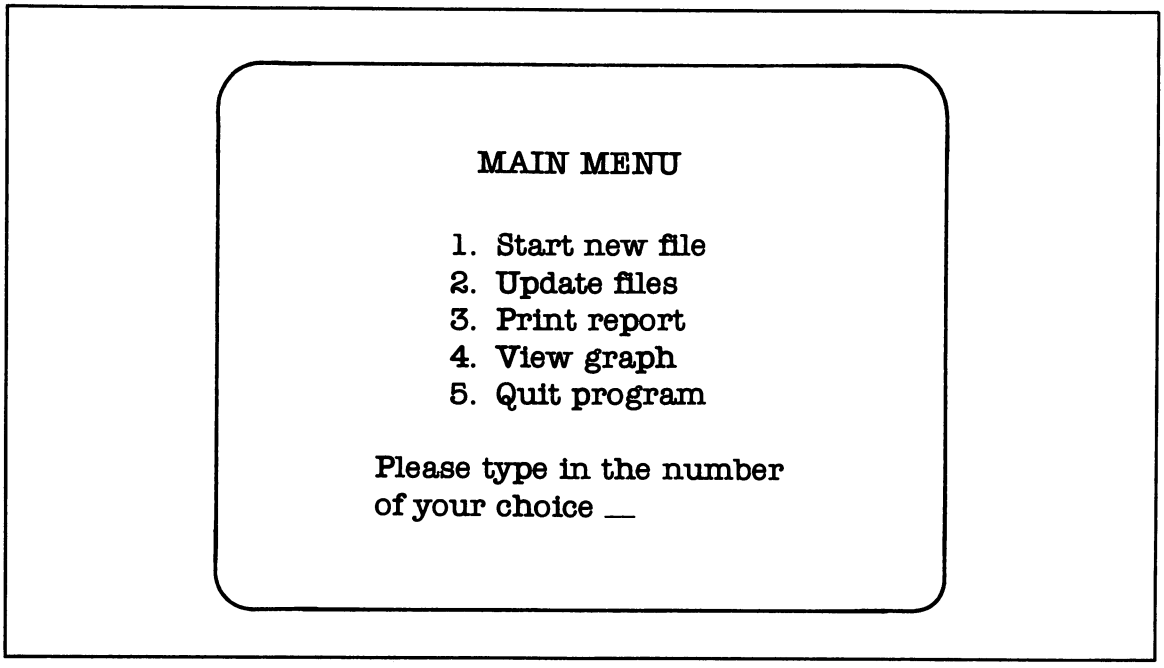


Fig. 1-3. A menu is often used to provide the user with a method to control the functions performed by a computer program. The user selects the desired menu option, and the program then performs the indicated function.

User Documentation

User documentation is information that tells the user about your program. It may come in the form of a user's guide, as help screens or other informative information within the program, or as a combination of both.

As someone who has experience with microcomputers, you know that much of the user documentation provided with programs is inadequate. Often it is incomplete, confusing, inaccurate, or all three. To make matters worse, much user documentation is badly formatted and does not even look good. How many programs do you have whose documentation consists of a semireadable photocopy with a staple in one corner? If you are sensitive to grammatical and spelling errors, you may wonder whom the publishers hire to write their manuals.

As in most things, there are good ways and bad ways to prepare user documentation, and, again as in most things, the bad ways far outnumber the good. Since no one has ever devised a specification for what to put in user documentation, diversity and inadequacy are the rule, rather than the exception. There is hope, however.

There are ways to decide what combination of written and within-program documentation your program should have. It is also possible to prepare a general outline that tells what a user's guide should contain. By following a few simple rules, you can devise a user documentation package that is adequate, if not exemplary. It is certainly possible to do better than many program publishers have done.

How important user documentation is to you depends, of course, on who will use your program. If you are the only one, then it is not very important, since you already have this documentation in your head. If your program will be used by others, however, user documentation is very important. No matter how great your program is by itself, without adequate documentation to explain it to others, it is like some alien spaceship that just landed in your yard.

You know, you open the hatch, walk in, and observe translucent rods, flashing displays and hear

a strange humming sound. You decide to fly it downtown to pick up a pizza at Pernicano's.

How do you get it off the ground? How do you steer it to your destination? How do you get safely home? How about if the aliens had thoughtfully left you a user's manual (preferably in English, and with lots of nice illustrations), or had a little screen that said, "Touch me for help."

User documentation is discussed in greater detail in Chapter 10. (It may or may not help you prepare user's manuals for spaceships.)

MAKING YOUR PROGRAM FRIENDLY TO PROGRAMMERS

Programmers are usually concerned with debugging their programs, maintaining (that is, making minor modifications or customizing) their programs, and sometimes with overhauling them or developing completely new programs based on pieces of the old. Several factors affect how easily these things can be done.

One of the factors is program size. If a program is small, figuring out what it does, debugging it, or upgrading it is usually easy. As a program gets bigger, however, at some point it becomes too complex for you to encompass in your mind. There is no formal way to decide when this point has been reached, although experienced programmers have a good idea. Anyone who has developed a BASIC program that is more than, say, 50 lines long, knows that it can be difficult to unravel what each line does.

A second factor that affects a programmer's understanding of a program is how it was designed, and what principles, if any, the original programmer followed during design. For example, it is usually easier to understand a program that is structured into modules that are identified with remarks in the code than one that is unmodularized and without remarks.

If you believe in writing programs that are friendly to other programmers, then you must make your programs easy for other programmers to understand. Of course, you may be the only programmer who ever looks at your code. Is there any point in worrying about friendliness then? If you have total recall, the answer is no. Since most of us suffer from imperfect memories, however, it is a good idea to design all of our programs as if they had to be understandable to other programmers.

A third factor that affects how friendly your program will be to programmers is how well you document it. Documentation written by programmers to explain how a program works is called *system documentation.* It consists of remarks within the code and various items of written documentation: written explanations of the subprograms; tables of functions, arrays, and variables; descriptions of subroutines; file descriptions with record layouts; and so forth. All of this documentation is intended to explain how the program works in enough detail that a programmer can make sense of it. Based on this understanding, the programmer can then correct an error, modify the program, or do something else that requires changes to the program code.

It is a simple fact that without adequate system documentation, a BASIC program that is more than a few lines long is usually incomprehensible to another programmer. Actually, with time, incomprehensibility becomes a problem even for the program's original author. Because our memories are faulty, we forget things. If we do not document our programs, in time our own creations become mysteries to us.

Some programmers do not see the point of writing system documentation. This may be due to their desire to keep their programs a bit mysterious. That way no one else can understand them, modify them, or threaten anyone's job. We have all heard or read about some programmer who was the only person in the world who knew how a particular program worked and was still called on, years later, to keep it running. When this happens, it is not always the programmer's fault, of course, but often it is. If you like to keep secrets, then ignoring system documentation may be your metier. Since you are no doubt more enlightened than this, however, you can surely see the value of such documentation.

You must decide whether or not you need such documentation and, if so, how much you need. If

you write a short program, you can often safely forget about it. With longer ones, you need it. Also, if you are the only person who ever analyzes your code, it is less important than if others also do.

Whatever your decision, it is well to design your programs so that they are readable. Techniques for doing so are described in this book. The last section of this chapter, for example, describes how to modularize a program, and Chapter 3 provides a model of a modularized program. The later chapters in the book provide many concrete examples of program modules.

System documentation is something you should understand well before you develop a program. Ideally, you should prepare this documentation as you code. The subject is covered in greater detail in Chapter 3.

GETTING CONTROL OF YOUR COMPUTER

You are probably very familiar with your C-64's BASIC, DOS and printer commands. They are covered in Commodore's documentation, as well as in many books and magazine articles that have been published since the C-64 was introduced. Since there is considerable documentation available, what does this book offer that is new?

First, some of the things that the C-64 can be programmed to do—such as linking, or *chaining*, between programs—are not explained well in documentation. The C-64 also has some features that are not documented at all.

Second, the C-64's BASIC, DOS, and printer control commands have several limitations that handicap the serious programmer. If you come to the C-64 with experience on other computers, then you know what these limitations are. If most of your experience is with the C-64, then you may be under the impression that your C-64 is limited to the BASIC, DOS, and printer commands that Commodore supplied with the hardware. Not so. In fact, many of the C-64's shortcomings can be overcome by using special techniques. These techniques will, however, make more sense to you if you understand the C-64's limitations at the start.

You know that you are in control of your computer when you have effective and reliable ways (usually subroutines) to do such things as:

- Display information on your video display.
- Collect data entries through the keyboard.
- Select and design the method of program control.
- Chain (that is, link) separate subprograms together.
- Design files and handle your disk drive.
- Control your printer and display information on it.

The key to this control is modularization. A module may be of any size, but usually it is a few lines long. Subroutines are modules. Some modules include several subroutines; and some modules are even larger.

One of the most important ways to gain control of your computer is to tailor how you think about what a program is. Experienced programmers tend to think of programs as collections of modules, rather than as collections of BASIC statements. They do not usually write programs line by line, but rather, build as much of a program as they can from modules in their module and subroutine library. This method is faster, easier, and more efficient than slogging through the business of building a program line by line.

Undocumented features are discussed throughout this book and not collected under one heading. Program modularization is covered later in this chapter. In the rest of this section, we will focus on the limitations of the C-64's BASIC, DOS, and printer commands. One of my objectives in this section is to make you aware of the C-64's shortcomings. Many of my comments will be negative. Please do not interpret them as criticism of the C-64's design. I could say many more good things about it than bad, but that is not my purpose here.

Limitations of 2.0 Level BASIC

Commodore's 2.0 level BASIC appears to have been optimized for graphics and sound and was short-changed in its repertory of advanced commands. This BASIC is not as powerful as the 4.0 level BASIC used on larger Commodore Business Machines.

Error Handling. One of the most serious shortcomings of 2.0 level BASIC is its lack of an error-handling statement. Most advanced BASICs have an ON ERROR GOTO statement. This statement is usually inserted early in a program and followed by a line number. If an error condition—syntax error, illegal input, DOS error, etc.—occurs during program execution, control jumps to the specified line, rather than interrupting the program and halting execution. Thus, control could be sent to an error-handling routine, which displays the type of error and line number to the program user. Later, control could be returned to, say, the main menu and the user could continue with the program. The C-64 lacks this statement, or anything like it, so most error conditions cause the program to crash.

The C-64's DOS allows you to write an error-trapping routine for handling error conditions that occur while attempting to read or write text files. The required procedure is most charitably described as involved. Your program must OPEN and read the error channel each time it attempts to read or write a file. If an error condition occurs, the error channel transmits a value other than 0. The program must collect and interpret this value, close the error channel, close the file, exit the file read or write routine, and then tell the program control what program line to jump to. This technique prevents program interrupts during file reading or writing, despite its cumbersomeness. Of course, it only takes care of text file read/write errors. If a syntax error occurs, if you attempt to LOAD a new program that is not on the disk, or if the user finds a hole in one of your input routines and manages to produce some sort of error condition, then put on your parachute.

Program Chaining. As a program grows in size, it is often wise to break it into subprograms which are loaded into memory as needed. Usually it is desirable to transfer the values of variables from one subprogram to the next. This is possible if the language being used has a convenient way of chaining from one program to the next.

C-64 BASIC permits one program to LOAD and start executing another, but transferring variables is another matter. The Commodore 64 User's Guide states that all you have to do is LOAD the second program from the first. For example, include this as the last statement of Program 1:

```
20000 LOAD "PROGRAM 2", 8
```

True, you can get the second program up and RUNning this way, but whatever variables and strings you attempt to transfer will usually turn to garbage. There is a way to chain and keep the integrity of your variables (see Chapter 7), but it involves several steps and is much more difficult than using the simple "CHAIN" statement that is found in many BASICs.

Verifying Files. Before you attempt to chain from Program 1 to Program 2, it is important to be sure that Program 2 is on the disk that is in the drive. This presents no problems if everything—all programs and data—is on one disk. If not—if your program is too large to fit on one disk, or if you use separate program and data disks—then there is the possibility that your program may attempt to read a file that is not present on the disk. This causes an error condition and may halt the program.

Many BASICs have a VERIFY statement that can be used within Program 1 to determine whether or not Program 2 is on the disk before attempting to chain to it. If so, the chain is executed. If not, Program 1 presents a message telling the user to insert the appropriate disk. Verification also works with text files, and a similar procedure can be used before attempting to read or write them.

Unfortunately, the C-64 lacks this type of VERIFY statement. The C-64's VERIFY statement is used to test whether the program currently in memory is the same as the one whose name you gave following the statement. This does not help when attempting to link two different programs.

There is a way to verify files with the C-64, but it requires ingenuity. The details are contained in Chapters 7 and 8.

Screen Control. The C-64 has a good assortment of the BASIC statements required for generating screen displays. Your C-64 can display information in standard text format, uppercase and lowercase, normal or reverse video, and in color. It

has powerful graphics. One missing feature is the ability to present flashing characters. Flashing can, however, be done with a BASIC subroutine described in Chapter 5.

Cursor control in the X and Y coordinates of the screen is possible, but cumbersome. Ideally, BASIC should let you move the cursor freely about the screen in both the vertical and horizontal directions. Models of convenience are Applesoft BASIC's VTAB (vertical tab) and HTAB (horizontal tab) statements. These statements permit Apple users to move the cursor to a particular vertical and horizontal position on the screen by using the VTAB and HTAB statements followed by the vertical and horizontal coordinates, respectively. For example, the following statements move the cursor to vertical coordinate 10 and horizontal coordinate 20:

```
VTAB 10
HTAB 20
```

These statements make it easy to move the cursor around the screen to generate displays and character graphics. Such cursor control frees you from generating a screen one line at a time, top to bottom. Instead, you can generate any part of the screen you want, in any order.

You can create C-64 routines that do almost the same thing as VTAB and HTAB, but they are slower and have limitations. Chapter 4 describes such C-64 BASIC techniques, and provides a set of assembly-language subroutines that are simple to use and that increase the C-64's cursor control.

Another thing that comes in handy during screen generation is the ability to clear (that is, erase) a particular line or range of lines so that new information can be printed over in the place of old information. C-64's BASIC allows you to clear an entire screen, but nothing less. Single lines can be cleared by printing strings of spaces at the desired location. Under some conditions, however, this procedure has undesirable side effects. Again, Chapter 4 describes the problem and a workable solution.

Printer Control. The VIC-1525 printer is an inexpensive, slow, good quality printer. The BASIC statements required to control it or other C-64 printers are unnecessarily complex, limited, and poorly documented, however. The printer control commands make printing information at a particular tab position no mean feat.

When you attempt to move beyond simple things like printing names at the left margin or entire lines of text at once, you run into difficulties. For example, if you want to print several columns of numbers, with each column aligned on the decimal point, C-64 documentation may lead you to the conclusion that it is all but impossible. Actually, it is not impossible, but it is not easy. Details are given in Chapter 9.

Program Development Utilities. The C-64 has a nice line editor. You can insert characters into or delete them from a line very easily, and the cursor does not have to be at the end of the line when you press the Return key for the line to enter memory. The keyboard is also nice. The keys are sculptured; they have a nice feeling, and the slope of the keyboard is about right.

If you look beyond these features, which are obviously very helpful to a programmer, you discover some problems when using the C-64 for program development. Important utilities and commands that a programmer needs in order to develop a program are not available in the C-64's BASIC or DOS.

For example, take a simple thing such as deleting a range of lines. During program development, you often need to delete a section of code containing more than one line. Most BASICs permit you to do this with a DELETE statement or something similar. You type in a statement such as DELETE 1010-2136 and the computer removes all lines between 1010 and 2136 from the program. The C-64 has no such statement; so you must delete each line separately. This is enough to put off any programmer who values time. Clearly, you need some way to simplify line deletion.

In addition, during program development you need several other utilities:

- Renumber—Changes the numerical increment between line numbers and the starting line number.

- Merge—Combines two separate programs into one.
- Search and Replace—Finds and replaces a particular character in a program with another character.
- Line Cross-Reference—Tells what lines in a program are called from other lines with GOTOs or GOSUBs.
- Disk-Disk Copy—Copies the content of one disk to another.

Utilities exist for doing these things, but do not come with your computer as delivered. Finding the required utilities takes a bit of research. My recommendations appear in Chapter 2.

Program Readability. The C-64 uses graphic symbols to represent many of its statements. The heart, for example, represents the command to clear the screen. Several dozen symbols are used, and a program listing with more than a few is unreadable to all but the most experienced C-64 programmers. For a programmer, learning all of the symbols is a formidable task. Usually, it is easy enough to type in a program from a keyboard. When the keystrokes become parts of program lines, however, many of them change to symbolic form, making it difficult for a programmer to decipher the program. One solution to the problem is to avoid the quoted form of the statement and to PRINT the CHR$ equivalent (see Appendix B). For example, the statement PRINT CHR$(147) clears the screen just as if the shifted CLR/HOME key had been pressed and included between quotes. (CHR$ codes must also be memorized, but they are easier to look up in an ASCII table than symbols. They therefore make a listing more readable.) An alternate or additional solution is to use remarks in code to clarify statements that are unfamiliar and that may cause confusion. Program readability is discussed in greater detail in Chapter 3.

DOS Limitations

C-64 DOS has several shortcomings. Most of them can be gotten around in one way or another. Without knowing the tricks, however, you may find that working with the C-64's naked DOS is rather like attempting to play tennis with a badminton racket.

Reading the Disk Directory. Most DOSs have a command that allows you to display a directory which shows what files are on the disk. Common forms are the CATALOG or DIRECTORY statements. With the C-64, to display the directory, you must LOAD a file named "$" and then LIST it. Doing this erases whatever program is currently in memory. Going to all this trouble to read the directory—and losing the current program as well—is obviously very inconvenient.

Saving a Program. The command required to save a program is:

```
SAVE "@0:PROGRAM NAME", 8
```

Several things about the syntax of this command are irritating. First, unless you insert the @0, DOS will not let you SAVE the file. This may have been intended to prevent programmers from overwriting an old file with a new one; however, most programmers would probably prefer the extra risk if it would save them keystrokes. Another irritant is the requirement to put the program's name in quotation marks. Finally, there is the requirement to end the SAVE statement with a comma, followed by the number 8. Eight you recall, is the device number of the first disk drive. If you leave the ",8" off, DOS assumes you are saving the program to cassette. Since the default is to cassette, the C-64's designers expected the most C-64 users would use cassette, not a disk drive.

In short, you must type in a lot just to SAVE a program. The command syntax is hard to remember, and invites errors. Fortunately, there is a simple way to get around this, as discussed in Chapter 2.

Reading and Writing Text Files

C-64 disk drives have two great things going for them. First, they are cheap, at least in comparison with those of most microcomputers. Second, they have a respectable storage capacity—170 kilobytes.

Now, the drawbacks . . . If you have read the VIC-1541 Single Drive Floppy Disk User's Manual,

and made good sense of it, you deserve credit. The manual contains much useful information, but its presentation is often confusing, incomplete, and in some cases inaccurate.

After working with this manual for a while, consulting with your friends, and perhaps calling the Commodore hotline, it may have occurred to you that it is not just the manual that is bad, but the DOS itself. For example, the syntax used in routines to read or write text files is unnecessarily complex. There is also a fairly small upper length limit on random access files (254 characters).

The hardware also has some drawbacks: the slow, serial port of the disk drive makes file reading or writing a time-consuming process. The parallel interfaces used on many microcomputers are much faster, often by a factor of three or four times. The slow data transfer rate is particularly obvious during disk copying. If you have a single drive, and have copied disks with it, then you know that it takes several minutes to transfer the content of one disk to another. (One single-drive disk copy utility I have used takes more than 20 minutes and five disk swaps to complete the copy. It makes you see the wisdom of having two disk drives and making them copy on their time, rather than yours.)

Overcoming the C-64's Limitations

Now for the bright side! The C-64 has shortcomings, but there are ways to overcome most of them. To do serious programming, you must. This book—particularly Chapters 4 through 9—will tell you how. To give you an idea of what is coming, let us briefly preview some of the subjects covered in those chapters. Each of these chapters contains several examples of BASIC code to illustrate the discussion.

Chapter 4 covers screen design principles, planning, and content. It presents subroutines for cursor control, partial clearing of the screen, number formatting, and other aspects of information display.

Chapter 5 tells how to handle the process of taking user data entries. This includes error-testing, error messages, and user verification and editing of entries.

Chapter 6 describes program control methods and gives guidelines for deciding what type of control technique—such as menu, two-way choice, or typing in a program name—to use in a program. It also shows how to combine different control techniques.

Chapter 7 tells how to chain successfully between programs with the C-64.

Chapter 8 tells what you need to know to use text files. It covers file planning, sequential and random access files, reading the error channel, and how to develop file read/write subroutines that you can use in all of your programs.

Chapter 9 tells how to control your printer from within a program—turn it on and off, and move the print head where you want it. Guidelines are given for designing hard-copy reports.

The other chapters in this book are also important, but in a different way. Chapters 2 through 3 cover preliminaries—things you should know before starting program design. Chapter 10 covers user documentation. Chapters 4 through 9 cover the very practical and down-to-earth business of writing program code. As such, these chapters have many examples of actual code—especially subroutines. They will give you a chance to exercise your fingers as well as your mind.

PROGRAM DEVELOPMENT STRATEGY

This section presents a strategy for developing microcomputer programs. I call it a "strategy" because it does not consist of a rigid set of steps, but a general approach. Engineers and others have written detailed program-development procedures in the past, but no single procedure works well for everyone. Programmers have different personalities, talents, and working styles; one size does not fit all.

This strategy consists of a mixture of procedures, requirements to meet, and warnings. It is admittedly a hodgepodge, and if you look at any single part of it too closely, you will find plenty about which to complain. Actually, when you boil it all down, the resulting essence is a sort of program-development philosophy. As you will see, this philisophy places heavy emphasis on making

your program friendly to users and programmers, on planning, on complete documenting, and on careful testing of the final products. The place to begin this discussion is with two key aspects of design philosophy—top-down design, and modular design.

Top-Down Design

Top-down design is rather like the concept of truth: most people seem to agree that it is a good thing, and many scholars make their livings writing about it, but it is difficult to find a simple explanation of what it means. I will forego the technicalities here, and offer a practical, common-sensical way of thinking about it. I will start by offering two case studies of design approaches. You guess which one is top-down design.

In Case A, the programmer sits down at his computer and types in the first line of code. When he finishes that, he types in the second line, the third, and so on, until he has typed in the last line of code. Then he RUNs the program to see if it will work. When bugs pop up, he fixes them. He continues in this way until the program works to his satisfaction.

In Case B, the programmer sits down and writes a design plan. First, she decides what her program's objectives are. Then she decides what functions her program must perform, and decides what modules are required to perform each function. She then goes to her subroutine library and looks for pieces of old programs to use in the new one. She sits down at her computer and develops the first module. She tests it, debugs it, and sets it aside. She then develops, tests, and debugs the next module. She tests how well this module works with the first module. She continues in this way, module by module, until the whole program is developed.

It is not hard to guess which designer used top-down design. She was the systematic one. Her design approach was characterized by these features:

- She started by defining the program's high-level objectives.
- Next, she broke the program down into modules.
- She developed each module separately, thoroughly testing and debugging it before continuing.
- As she developed new modules, she tested their interface with old modules.

The top-down approach forces a program to be developed and debugged systematically in a series of logical steps (Fig. 1-4). This contrasts with the more haphazard approach of a programmer who sits down and simply starts writing program code.

Top-down design is increasingly important as a program grows in size and complexity. If you write very short and simple programs, then you can safely ignore it. If you want to do something ambitious in a program, however, then use top-down design. It increases your power as a designer greatly.

Modular Design

What is a program module? It is a segment of code that performs a specific function. A subroutine is a module. Modules exist on higher levels, as well. Often a module includes several subroutines, and performs a high-level function, such as creating a set of data files. In fact, there is no upper size limit to a module.

What is unique about a module is that it does something more sophisticated than a single statement of program code. In a sense, a program module is to a programming language what a high-level language, such as BASIC, is to the machine language of the computer. Just as programmers find it easier to work with high-level languages than with individual bits and bytes, most skilled programmers find it easier to work with program modules than with individual statements of code. Obviously, you cannot ignore the BASIC language or DOS, since your program is composed of it. It is a good thing to start thinking of your program not as so many lines of code, but on a more global level—as modules.

The starting point in modular design is the programmer's subroutine library. One thing nice

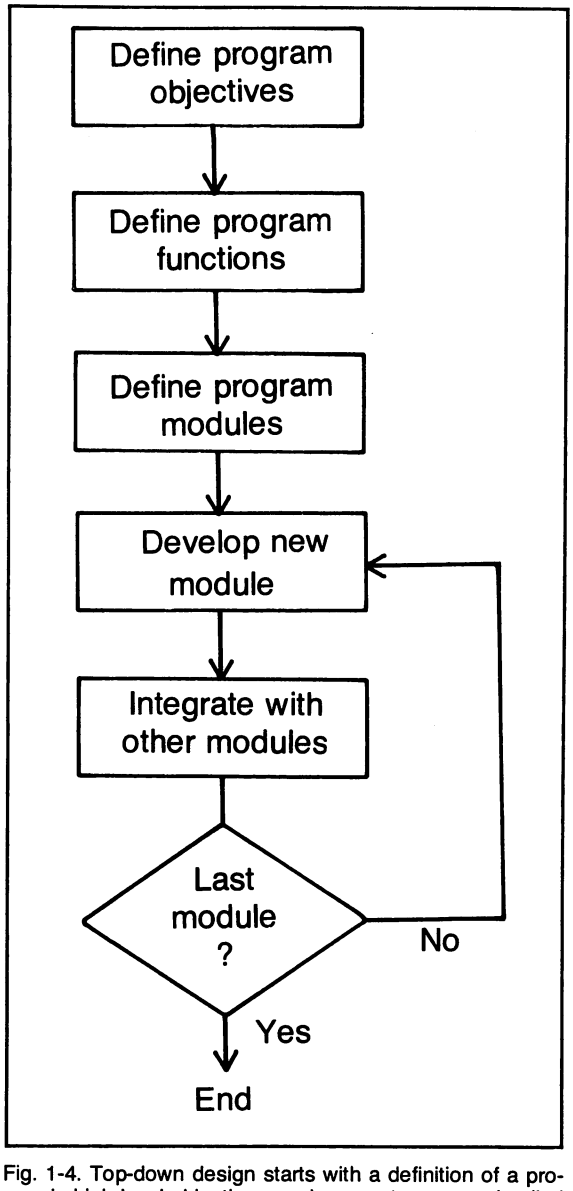


Fig. 1-4. Top-down design starts with a definition of a program's high-level objectives, and moves to a more detailed definition of the program's modules and their interconnections.

about having a good subroutine library is that it makes it easier for you to design a new program. You can take pieces of the old and put them into the new. A good subroutine library consists of subroutines that have been carefully tested and debugged. You can use them without testing them because you already have and know that they work properly. A large part of a new program may consist of these subroutines. Creating the program becomes a matter of writing the control structure (that is, subroutine calls) and designing any unique input or display routines.

One other advantage of this approach is that your programs begin to look similar. Since they are all built of many of the same modules, you can go from one to another without a lot of reorientation. You also have a good idea of where problems are likely to occur, and find it easier to correct them.

Critics may argue that this approach to design is not very elegant. The reason is that it does not solve each design problem from scratch and look for the most efficient solution. Instead, it concentrates on ways in which the new problem is similar to ones that have already been solved and then uses the pieces of the old solution in solving the new problem. However elegant or inelegant the approach, it has two enormous advantages: (1) it allows rapid program development, and (2) it minimizes errors.

Obviously, this approach is not for every programmer, or for every program. In most cases, however, and with most programmers, it works very effectively.

Now that you have a taste of design philosophy, it is time to look at the steps in the design strategy. These steps are illustrated in Fig. 1-5 and are discussed here. Remember, there is nothing rigid about this strategy, and you may modify the order of the steps, eliminate or add steps, or make other changes that you think necessary. If you do make such changes, however, be sure that you do not sacrifice the strategy's emphasis on the user.

Step 1: Start with Users

The place to start any design is by knowing who will use your program. You must tailor your program to this audience. You must design displays they will understand, test for the types of errors they will make, and provide a method of program control they can use effectively. Later on, you must develop user documentation that will help them

make the most of your program.

Know who these folks will be at the beginning. Do not wait until after you write your program. Then it will be too late or too expensive to make changes.

Step 2: Plan Your Displays

Decide what will go on your displays before you write your program. Displays are what the user will see. Since the main idea of most computer programs is for the user to interact with the computer through a display, the program should be designed backward from these displays. It is common but wrong to develop a program the other way around—that is, design displays based on the program that has been developed.

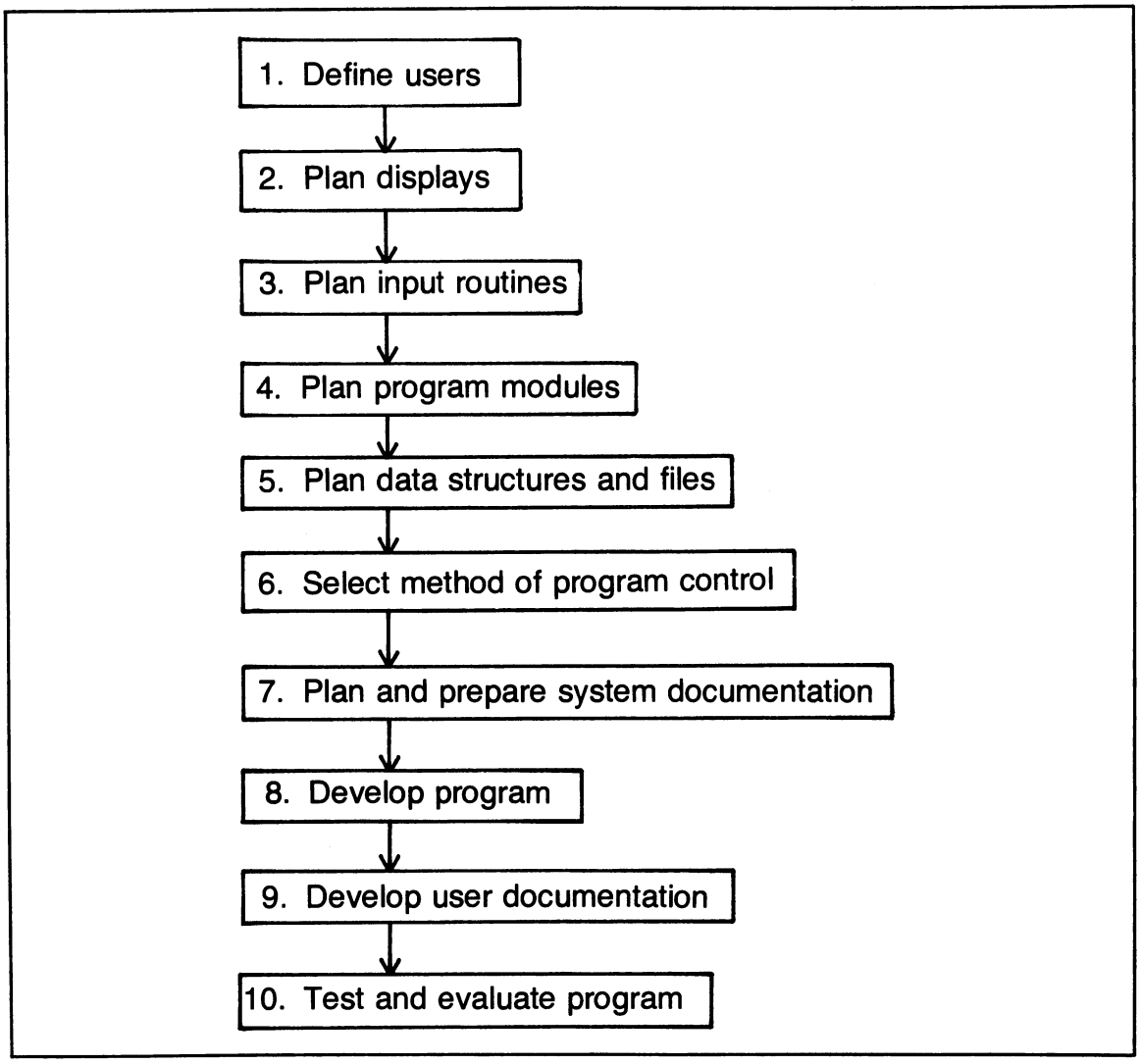


Fig. 1-5. A top-down program development strategy. The steps in this stretegy place heavy emphasis on the user—starting by defining the user, the displays the user will view, and the program's input routines. The last two steps also stress the user—developing user documentation and having the user test and evaluate both the program and its documentation.

You do not have to plan every last detail of every display at the start, but design the displays as completely as you can. Make and show display mockups (on paper or on the video display) to your target audience, if you can. If you do this, then you have three or four steps up on the poor programmer who completely develops a program, only to discover that users cannot make sense of its displays.

Step 3: Plan Your Data-Entry Routines

Your computer will talk to the user through its displays. The user will talk back to the computer through the keyboard or through some other input device. Together, computer output and user input comprise what some people refer to as the "human-computer interface." It should be completely designed before you actually begin to code your program. Decide what data must be entered and then plan your input routines—the prompts, error messages, and verification and editing routines.

Step 4: Plan Your Program Modules

Decide what modules your program will include. Start at the highest level and work your way down. An example of a high-level program module is a data-entry program that is used to collect data from the user.

Break each module down into submodules, until you get down to the level of the subroutine. Then go to your subroutine library and select subroutines to use in the submodules.

Overall, you will need subroutines to handle display (output), user input, program control, program linking, file-handling, and printer control. (Conveniently and coincidentally, Chapters 4 through 9 contain many subroutines for these purposes.)

Step 5: Plan Your Data Structures and Files

Plan your data structures—the way you will represent data within the program using integer, real, and string variables, and arrays. Plan your files and lay out the records that are required.

Step 6: Decide on a Method of Program Control

Decide how your user will interact with the program to control its operation. That is, decide whether the program will be menu-driven or use some other control method. Design the menus or other control screens.

Step 7: Plan and Prepare System Documentation

Begin developing system documentation as you plan the program. Planning program details such as data structures and record layouts is an important part of system documentation.

Prepare additional documentation as the program evolves. As much as possible, develop system documentation concurrently with the program, not at the end.

Step 8: Develop Your Program

Start by developing the primary module in your program. In a program that manages a data base, the primary module may be one to read or write files. In a program that uses extensive graphics, the primary module may be one to generate displays. Pick the module that is central to the operation of your program and develop it. Write system documentation for it. Then, develop the next logical module and its documentation. When this module is done, link it to the first module and test the two together. Make the interface between the modules work the way it should.

Next, go to the third module, link it to other modules, test the interface, and document. Continue in this manner until you have developed, integrated, debugged, and documented all program modules.

Step 9: Develop User Documentation

Develop a user's guide that tells your target audience how to use your program. Your user's guide should be comprehensive and describe every essential aspect of your program, including information on system setup, a tutorial for the new user, and reference information for the experienced user.

(Chapter 10 contains a detailed description of its content.)

In addition to written documentation, create the help screens and other internal documentation that your program requires. They should support the written documentation, and be optionally selectable by the user.

Step 10: Test and Evaluate Your Program

Testing goes on throughout program development. When the program is finished and user documentation has been prepared, try it out on its target audience. Train them in its use, give them the user documentation, and then have them use the program to perform specific tasks that you define. See how well users perform and how well they accept—that is, like—your program. Do not be surprised if they discover program bugs that your own tests missed.

Following these tests, revise your program to correct its shortcomings and maximize user acceptance.

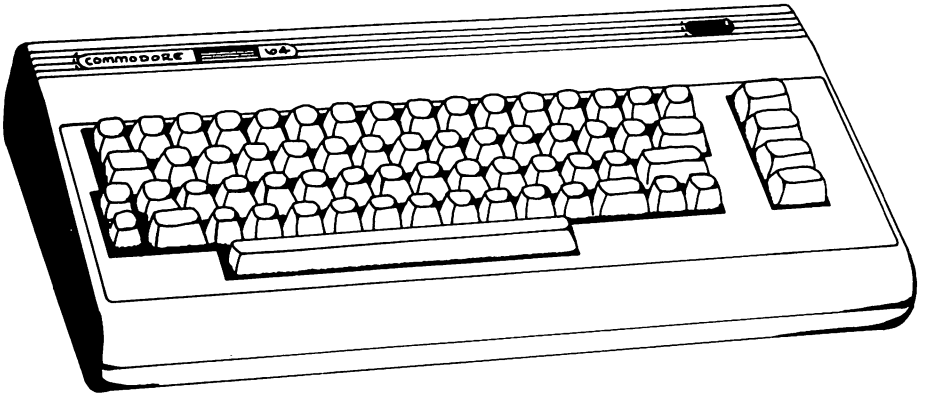
A Final Word

In reality, I know that no programmer will ever follow this formula exactly in designing and developing a program. As noted, this is more of a design philosophy than a rigid formula. You should feel free to modify it to suit your working style and programming task.

Pay close attention to the essentials, however. This is a very user-oriented approach, with the user being consulted during the early part of design and acting as the evaluator at the end. Also important is the approach's emphasis on modularity, and systematic, step-by-program development.

I will fill in the details later in the book. Before continuing, I suggest that you store this program development strategy up in your high memory area where you can gain ready access to it, without disrupting your normal flow of control.

Chapter 2



Getting Started

This chapter describes several ways to increase the power of your C-64 as a program development tool. It begins with a discussion of hardware—the C-64 itself, video monitors, printers, and disk drives. The discussion also covers the important subject of work-station design, and tells what types of files (paper, not computer) you should use to keep track of your programs and listings.

The second section of this chapter covers software. It describes several ways to soup up your BASIC and DOS, and recommends various program development utilities. The third section identifies and describes several books and magazines about which you should know.

HARDWARE

You presumably have a Commodore 64 computer and know all about it. These computers are all, more or less, the same—light brown with 66 keys and four function keys on the outside, and with 64K or RAM and 20K or ROM inside. They can generate 16 colors for characters, screen, and border; have uppercase and lowercase characters, high resolution graphics, graphic characters, and a sophisticated music synthesizer; and their accompanying peripherals—disk drive, monitor, and printer—can be plugged right in. Their peripherals are smart, and require less computer RAM, leaving more memory—38,911 bytes, to be exact—for your program and its variables. In all, the C-64 is an impressive package—powerful, convenient to use, and inexpensive.

The C-64 Computer

On a more pedestrian level, here is a hint for improving your C-64. If you sometimes have the experience of seeing garbage appear on your screen because your fingers have lost their place on the keyboard, there is an easy way to solve the problem. Get a tube of epoxy glue, put a tiny bead of it onto a pencil point, and place it in the middle of the "D" and "K" keys. Let it dry, and then repeat a few times. The hardened bump will provide a tactile cue that tells your fingers where the D and K keys are

and keeps them from getting lost. Give the same treatment to other keys you use frequently without looking at —selected number keys, Insert/Delete, and Cursor keys. Simple things help.

Video Display

The C-64 has a built-in modulator to drive a TV set. Most C-64 owners hook their computer up to a color television set through the C-64's TV Connector and are perfectly satisfied with the quality of the picture they see. This is fine for anyone who is not critical about picture quality and who uses the display for less than, say, 30 minutes or so each day. Display quality is largely a function of how good the TV set is. If it is a cheap set, or an old castoff that has lost its contrast, alignment, and resolution, then it will probably cause eyestrain.

Some people use black and white television sets on their C-64s. The picture quality on these sets is slightly better than that on a color set, but without the advantage of color. If you must choose between a color TV and a black and white one, get color. The advantages of color outweigh the slightly poorer picture quality.

If you spend a significant amount of time using your C-64, however, get a video monitor, not a TV set. Monitors produce better pictures than modulator-driven TV sets. The C-64 has an output port that bypasses the modulator and sends the signal directly to the picture-generation circuits inside of the monitor without modulation in the computer and demodulation in the TV, and there is less interference and signal loss this way. When you use a modulator to drive your TV set, the computer's output must be modulated and translated into an RF (radio frequency) signal that your TV set can treat like something coming over Channel 3 or 4. (TV sets are not smart enough to know the difference between your C-64's modulated output and the "Tonight Show".)

There are several good-quality color monitors that will work with the C-64, and their main difference is price. Among the leading manufacturers of such monitors are Amdek, NEC (Nippon Electric Company), and Taxan. These monitors are widely available, often discounted, and of good quality. Their manufacturers sell both composite and RGB (red-green-blue) monitors. Your C-64 is not equipped to drive an RGB monitor. You must use a composite monitor with it. While RGB monitors produce better pictures than composite monitors, they are much more expensive, generally selling at about twice the price. The Commodore model 1701 or 1702 video monitor is relatively inexpensive—less costly than most color monitors or color TV sets—and produces a good picture. Better monitors are available at higher prices, but, in my opinion, they are not worth the extra cost. If you happen to have or prefer some other monitor, you can obtain a coaxial connector cable from Commodore that will allow you to connect a standard composite color monitor to your C-64.

There is still another choice—the monochrome (single-color) monitor. For extended use, monochrome monitors generally cause less eyestrain than color monitors. If you intend to work long hours before the screen, and color is not essential, give serious consideration to a monochrome monitor. If you decide to get one, be aware that the size and quality of the monitor are more important than its phosphor color. The three most common phosphor colors available on monitors are white, amber (light yellow), and green. The eye is slightly more sensitive to green when there is low ambient lighting, and so if you intend to use your monitor in a semi-darkened area, green is best. If you will use your monitor in brighter surroundings, then white or amber are better.

Whatever you decide to get—color or black-and-white TV set or color or monochrome monitor—get one that has a screen diagonal of at least 12 inches. A smaller screen than this is hard to read and not good for extended use. A larger screen is unnecessary and awkward to use if it is at a normal viewing distance (20 to 24 inches).

Printer

You need a printer to do serious programming. There are two reasons for this. First, a printer permits you to generate a hard-copy listing of the program you are writing. The listing is important during program development because you can usu-

ally find what you are looking for more quickly and easily with a hard-copy listing than by going through the program screen by screen on your video display. The listing is also a permanent record of your program that you cannot mistakenly erase, overwrite, or damage with a few careless keystrokes. Although careful working practices reduce the likelihood of such accidents, they occasionally occur anyway. Many programmers have been saved by the hard-copy listing of the program that they kept on file.

Another reason you need a printer is to provide hard-copy output for the programs you develop. Not all programs generate reports, but many of the serious ones do. A word processor, data base manager, or financial analysis program without hard-copy reports is unthinkable.

What type of printer should you get? Commodore manufactures several printers that will work with the C-64, and you can interface those of other manufacturers as well. The standard printer for the C-64 is the VIC-1525 graphics printer. This printer has been superseded by the VIC Model 801, which is packaged differently from the VIC-1525, but is similar. (Throughout this book I will focus on the VIC-1525, but the discussion applies also to the model 801 and other printers that use the same command set as the VIC-1525.) Commodore also sells more sophisticated and faster dot matrix printers.

The VIC-1525 is an impact dot matrix printer that uses standard 9½-inch tractor-feed paper. It uses a 6-×-7 dot matrix and can print letters, numbers, symbols, and graphic characters. The print head is dot-addressable, and can print point graphics. Print speed is a leisurely 30 cps (characters per second). The printer is sold by Commodore with the Commodore label, but it is manufactured by a Japanese company which sells the same printer under its own name.

The VIC-1525 uses a special serial interface and plugs directly into the C-64. It works with the C-64, but not with other computers. If you have a nonCommodore computer beside the C-64, or might get one, it is wise to get a printer that will work with both machines. The trick is to get a parallel interface for your C-64 that will drive a printer other than the VIC-1525. You can buy a Japanese printer that is identical to the VIC-1525 for about the same price, and it will usually come with a parallel interface so that you can also use it with your other microcomputer.

Several manufacturers sell parallel printer interfaces for the C-64. They cost less than $75, and plug into the expansion slot in the back of the C-64, where they permit you to plug in any printer with a standard (Centronics) parallel interface. Manufacturers of these interfaces include Cardco, Skyles Electric Works, Richvale Telecommunications, and Batteries Included.

You may want to consider a printer that is a little faster or fancier than the VIC-1525. If so, good alternatives include the Okidata, C. Itoh, and Epson—these companies make reliable and popular printers which offer greater speed and more features at only a slightly higher cost than the VIC-1525. If you want to do graphics on your printer, be sure that the interface and printer you get will do the job. Not all combinations are equivalent to the C-64 and its VIC-1525; check carefully before you spend money.

If you intend to do a lot of word processing, you may want to get a daisy wheel printer, one that prints fully-formed characters with a rotating "daisy" printwheel. Print quality is superior to that of a dot matrix printer. Several brands are available, ranging in price from a few hundred dollars up to several thousand, depending on features. These printers are good for producing typed copy, but if most of your computer output will consist of program listings or computer-generated reports, then save your money and get an impact dot matrix printer. They are less expensive, faster, more flexible, more reliable, and quieter.

Disk Drives

You need a disk drive to do serious program development work. This is true even if the programs you intend to develop will be stored on cassette. The cassette is a reasonable storage medium for users who have time and whose storage requirements are limited. Not so for the programmer.

You need the speed and flexibility that the disk storage medium offers. The C-64 disk drive is not notoriously fast. By the standards of most microcomputers, it is slow. It is, however, much, much faster than cassette.

Where C-64 disk drives are concerned, there are not many choices. The only ones that work are manufactured by Commodore. The standard drive is the VIC-1541 or VIC-1542. The two drives are functionally the same. Each has its own power supply and microprocessor, and you can hook up to five of them to your C-64. These drives are widely discounted and relatively cheap. Buy two of them, if you can afford it. Having two will be helpful in many ways:

- It makes disk copying much easier and takes less of your time.
- It allows you to keep different disks in different drives, and saves a lot of disk swapping.
- It permits you to develop programs that make use of more than one disk drive.
- It gives you a backup if one of your drives dies.

Commodore has had a quality control problem with some of its early VIC-1541 drives. (If you have one of these, and have been having problems with it, take it to your dealer for a checkup.) To correct its problems, Commodore changed the subcontractor that manufactured the drives. Whether this solved the quality control problem remains to be seen. It is no secret that many of the early C-64s as well as disk drives were shipped with defects. Commodore claims that their defect rate is no greater than that of other manufacturers, however. Whatever the case, it is certainly true that Commodore's reputation has not been helped by some of the press reports that have surfaced. It is also true that if you get a good C-64 and good disk drive, they give little trouble.

While the VIC-1541 drive is the most obvious choice for the C-64, there are other options. You can obtain an interface that will allow your C-64 to use the IEEE bus, and thereby gain access to the drives used with the CBM series of business machines. You may, for example, use the 4040 or 8050 drives. These are dual drives with greater storage capacity and speed than the VIC-1541. The 4040 can read and write VIC-1541 disks and is much faster and more convenient to use. Many programmers use it for developing C-64 programs, although it is not fully compatible with the VIC-1541. The area of incompatibility between the VIC-1541 and 4040 drives is in timing. Either drive can read the other's disks well enough, but writing to the other's disks may cause problems. Consequently, it is inadvisable to use one of these drives to write to disks developed for the other.

The 8050 drives use a different disk format. By expanding your C-64 with 8050 drives, you essentially create an inexpensive and more limited version of one of the CBM machines. Unfortunately, you also lose compatibility with the standard drive used with the C-64—the VIC-1541. A popular interface for using the 4040 or 8050 drives is the C-64 LINK, which is manufactured by Richvale Telecommunications. This device allows you to hook up the drives as well as other peripherals that use the IEEE bus. Similar products are manufactured by other companies.

Hardware Incidentals

If you live in an area where there are frequent power surges or drops, get a surge suppressor. If you live near a power station, this is not usually a problem. If you live on the outskirts—at the end of the power line—then you are likely to experience a greater amount of power variation. You know this is happening when the lights in your house dim momentarily, then brighten again, or you see other obvious signs of power fluctuations. These power variations can be very damaging to computer equipment. They produce spikes that can destroy electronic circuitry.

If you decide to get a surge suppressor, find one with a line switch on it and enough outlets for your C-64, monitor, disk drives, and printer. This allows you to use one power switch to turn everything on or off at once, instead of going through the numbers, component by component. It also saves the switches in your components. If you do not need a surge suppressor, get a power strip with a line

switch on it so that you can turn all of your computer equipment on or off with one switch.

Surge suppressors cost about $100. You can however, make your own for much less than this. If you like to do this sort of thing, review "Ciarcia's Circuit Cellar," in the December 1983 *BYTE* magazine for directions on how to do it.

Hardware Summary

The hardware I recommend for serious C-64 program development is shown in Fig. 2-1. It consists of a C-64 with two disk drives, Commodore color video monitor, and a parallel interface that is connected to a good-quality dot matrix printer. For some program development applications, you might substitute a monochrome monitor for the color monitor. If you do word processing, you can substitute a daisy wheel printer for a dot matrix printer.

Your Work Station

Work station is the label that ergonomics experts call the arrangement of furniture and equipment in a person's work space. If you use a computer, you have a work station, but you probably call it something else. There are good ways and bad ways to design your work station, and the good ways are not always obvious to common sense. If your work station is poorly designed, and you use it for more than an hour or so each day, then you may suffer from such symptoms as eyestrain, backache, or pain in your arms or wrists. The solution is to redesign your work station. Here are a few suggestions:

- Get a good chair, one that is reasonably firm and that provides good lower back support. If your feet do not touch the floor, get a foot rest or something else on which to set them.
- Locate your keyboard at about the same height as your elbow (arms at side). You should not have to reach up to or across a desk to touch it. The keyboard should be a few inches lower than the height of the

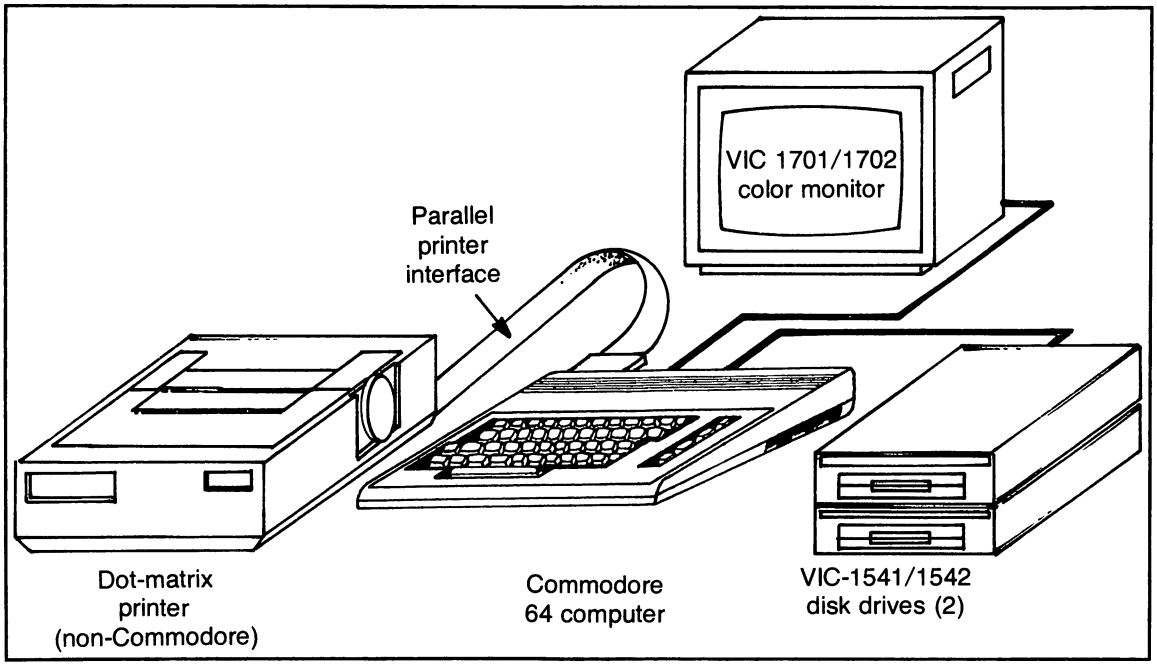


Fig. 2-1. The recommended hardware configuration for C-64 program development includes a color monitor, two disk drives, a parallel printer interface, and a good quality dot-matrix printer.

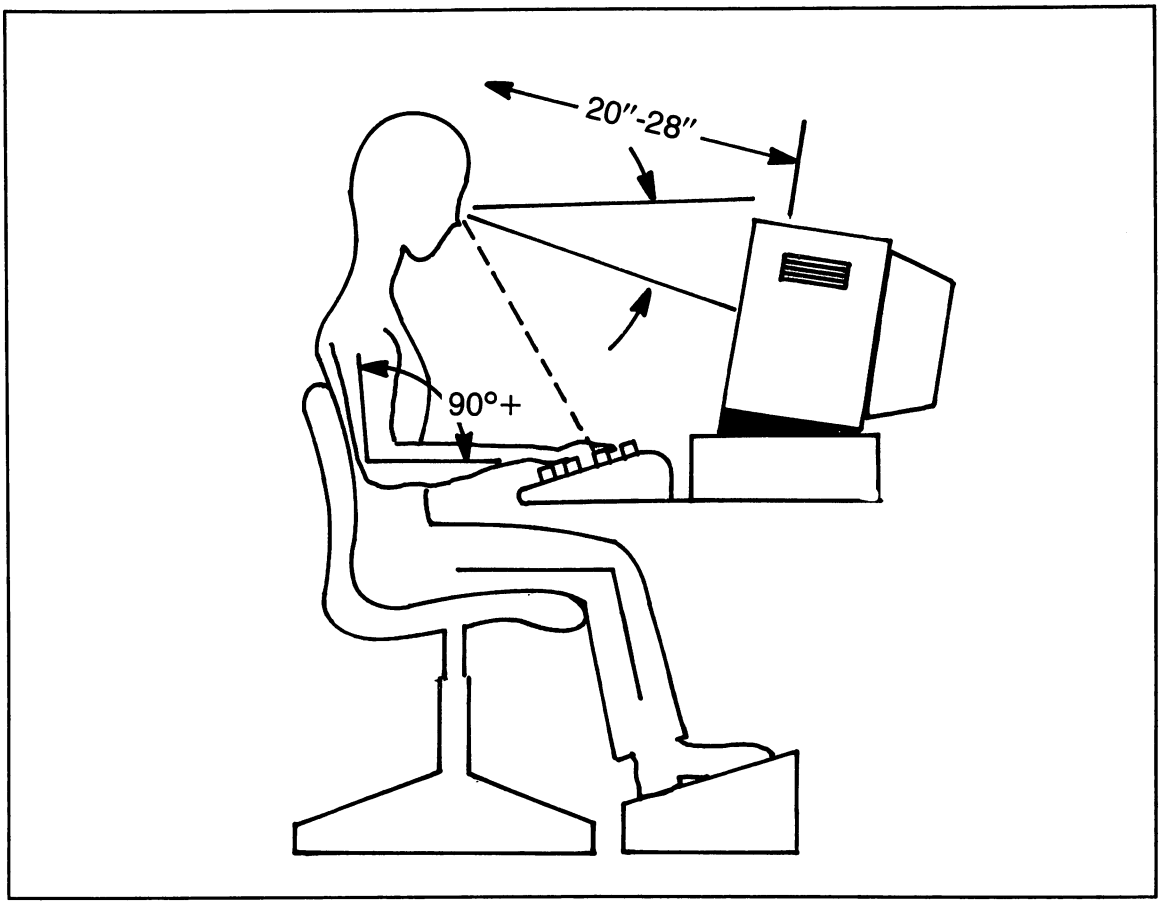


Fig. 2-2. Programmer's work station. It is important to have a chair with good back support and to locate the keyboard at about elbow height. The monitor should be about 2 feet away, at or slightly below the line of sight and positioned so it is out of the way of interference from lighting.

average desk. If you can raise your chair, this solves the problem, provided you can still get your feet to touch something solid. You should be able to move up close to the desk or table, and be able to get your feet underneath it.

- Locate your video display about 2 feet away, at or slightly below your line of sight when looking straight ahead. Most C-64 users find it best to set the video display on the surface directly behind their keyboard.
- Adjust the lighting so that there are no reflections from the screen of the video display or other bright lights directly behind or in front of it. Your eyes should adapt to the brightness of the video display, not a competing light nearby.

Figure 2-2 summarizes the requirements for a good work station. You do not have to go out and buy a $2,400 oak work station that has been hand crafted by a company with an organic-sounding name. Usually you can adapt inexpensive tables or office furniture to meet your requirements. If you cannot meet the recommendations just listed with the furniture and lighting that you have, however, get a catalog from a discount office furnishings outfit, take a trip to your local swap meet, make your own furniture, or look elsewhere to get what you need.

Files, Files, Files

Individual attitudes about filing cabinets and files differ, and probably have a lot to do with one's early childhood. If you believe that neatness and orderliness are important things, then you probably see the wisdom of filing cabinets and files. If you have a more casual attitude about these matters, then you may regard files as a bother that you prefer to avoid.

I would like to avoid taking a position on this issue, since it has so much to do with personal working style. It is impossible to be evenhanded about this since, in programming mattters at least, I tend to be a bit compulsive about such things as neatness and orderliness. A programmer who keeps his or her working materials—program listings, documentation, disks, and so forth—in a well-organized file is much better off than one who relies on memory of where things are to locate them. We all have friends whose desks look like dumping grounds for books, magazines, and debris, and often we regard them as brilliant eccentrics. Folklore says that such folks know exactly where everything is, and can find whatever they want, when they want it. In some cases this is certainly true, but I tend to regard such claims with skepticism. A programmer who treats working materials this way is courting disaster.

In programming, there is too little margin for error. Programs must be exact things, and the records and versions of them that must be kept exactly. Much of what a programmer does today is based on what was done yesterday—using pieces of old programs, troubleshooting a particular program module, writing a user's manual. In any of these or other related activities, it is necessary to be able to locate quickly the working materials we used previously. It is not good enough to be reasonably sure that what we have found is what we were after. We must be exactly sure, because we deal in an exciting discipline. Moreover, our time should not be spent rummaging through stacks of paper, looking through drawers or bookshelves, or performing other mindless drudgery.

What files do you need? Start with a flip file, one of those plastic boxes with a transparent cover that holds about 50 floppy disks. Put this on the table beside your computer and keep your frequently used programs in it. I suggest separate sections for utility programs, original and backup disks of the program you are currently working on, blank disks, and the other logical categories of programs that are relevant to you in your program-development work. The flip file is handy for ongoing work and for keeping commonly used programs ready at hand.

Once you complete a programming project, remove the program disks from the flip file and keep them in plastic storage boxes. I suggest that you get a half dozen or so of these to start, and use them for keeping the working copies of your programs. A plastic storage box costs less than $5—and cheaper in quantity—and holds about 20 disks. Label the boxes, put them on a shelf, and then pull them down when you need them. Colored storage boxes can be helpful. Use different colored boxes to store different programs, or to separate the disks used in your different computers.

If you do any significant amount of programming, get a file cabinet. Use it to keep your program listings, archival disks, documentation, information on hardware and software purchases, copies of articles, and other important material. Get file dividers and index tabs. Spend some time organizing your file. The test of your success is how quickly you can find what you want. You should be able to open a file drawer, locate the file tab, and pull what you want out in no more than about 10 seconds. If it takes longer, then you need better organization.

Organizing files—flip files, storage boxes, filing cabinets—in the way that I have recommended is extra work when you start. In the long run, however, it will save you work, confusion, and headaches.

Floppy Disks

What is there to say about disks? First, buy good ones, not cheap ones. Bargain basement disks are available, but avoid them. They will lose data and bring you grief. You do not have to buy the most expensive disks, either. Just stay away from the cheapies. The best way to buy disks is by mail

order. You can buy good-quality disks this way at a discount. If you want to give your local computer store business, buy the big things from them—computers, disk drives, printers, and so forth. The amount of extra service they can offer you when they sell you a box of floppy disks is miniscule.

Every programmer knows that taking care of disks is important. It does not take much more than common sense: keep them in a box when they are not in use so that they do not get dusty, do not handle them while you are eating french fries, keep your cat from walking on them. Also look out for the more subtle dangers. Pencil erasers are bad news; so do not spread them around where your disks are. Flea powder (if you have a pet), chalk (if you teach), diatomaceous earth (if you have a pool or aquarium), tobacco (if you smoke), and saliva (if you have a dog or a small child) all are dangerous.

How long do disks last? Nobody seems to know the answer to this question. Manufacturers are particularly cagey about it, and will not answer it directly. The answers I have received range from "about a year" to "forever." The conditions do not seem to matter. Along these lines, the best advice I can offer is not to use a disk for more than about a year. Keep a good supply of them on hand, and do not keep using the same ones over.

Disks for a programmer are like typing paper for a writer. They are working materials, necessary to do the job, and should not be treated as if they were made of some precious material. A good test of how cheap you are is how many blank disks you have. If you have fewer than 5 new, blank disks, then you are not being fair to yourself. Go out—no—call up a trustworthy mail-order firm—and buy yourself another 10. Better yet, buy 2 boxes. Get good ones.

SOFTWARE

There are three basic ways to soup up your C-64's BASIC and DOS:

- Do it yourself.
- Get a plug-in ROM card.
- Load in BASIC/DOS enhancements from disk.

The first option is open to you if you know where to find the appropriate assembly-language routines, can write them yourself, or both. This book offers some such routines, and other books and magazines offer more. Putting together a complete package of such routines however, is more trouble than it is worth, unless you like to do this sort of thing for recreation. One advantage of this approach is that it permits you to transport your BASIC/DOS enhancements on disk to another C-64. The other user does not have to have your plug-in ROM card or make use of someone else's copyrighted DOS routines.

The second and third options differ in how the BASIC/DOS enhancements are stored and in their convenience of use. With a plug-in ROM card, bringing up the additional commands is usually as simple as typing in a SYS command and then pressing the Return key. All the new powers are then brought to life. When the enhancements are on disk, they must be read from disk into memory, which takes a little longer.

A plug-in ROM card is the most convenient but least portable enhancement medium. If you develop a program that uses one, it may not run properly on a C-64 without the same ROM card. On the other hand, you can usually transfer disk-based BASIC/DOS enhancements to any disk and then use them on any C-64. Unfortunately, there is a complication if the enhancements are copyrighted. If you make many copies of them, and include them on the disk that contains a program that you intend to publish, then you must obtain permission from the copyright holder and perhaps sign a licensing agreement. Of course, if the copies are for your own use, the copyright factor does not matter.

There is a simple solution to both the equipment compatibility and copyright problems. In a nutshell, it is to use the BASIC/DOS enhancements during program development, but to design the program so that it will run on any C-64 that has 2.0 level BASIC.

This solution gives you the best of both worlds. You can use the BASIC/DOS enhancement features to speed up program development and de-

bugging, and also transport your program to another computer without worrying about copyrights or licensing agreements.

What are the drawbacks of this solution? Obviously, you cannot use certain enhanced BASIC/DOS commands in your final program. This book however, offers special techniques to make up for the loss. In fact, my feeling is that the greatest strength of most of the BASIC/DOS enhancement software is disk control, program editing, and debugging—that is, immediate execution commands that are not generally used within a program. Software packages differ, of course, in the features they offer. The best way to decide which one suits your needs is to investigate those that appear interesting. In the following paragraphs, I will give brief descriptions of several popular BASIC/DOS enhancements. My list is not comprehensive, but representative. Check out these packages, and review the advertisements in the magazines described in the last section of this chapter, to identify other BASIC/DOS enhancements.

Plug-In ROM Card

One of the most popular BASIC/DOS extension cards is Vic Tree™, manufactured by Skyles Electric Works, 231 E. South Whisman Road, Mountain View, CA 94041. The manufacturer claims that it provides Commodore BASIC 4.0 compatible disk commands which allow most Commodore PET/CBM programs to run on the C-64. It adds 42 new commands, mostly for disk control, editing, and debugging. The disk commands simplify such things as reading the disk directory, reading and writing files, copying disks (two drives are required), loading and saving programs, initializing disks, renaming files, and deleting files. Vic Tree has a powerful set of editing and debugging commands that includes automatic line numbering, trace, variable search and replacement, deleting a range of program lines, merging, renumbering, and screen control of program listings. Vic Tree also has an optional parallel printer interface.

As you are probably aware, the C-64 has been very popular in Europe and Canada and much good software has come from there. Richvale Telecommunications, 10610 Bayview, Richmond Hill, Ontario, Canada L4C 3N8, manufactures a wide range of software and hardware add-ons for the C-64 and other Commodore machines. Two of their products for the C-64 are of particular interest. The first of these is the RTC BASIC Aid, a programmer's development tool which adds 33 additional commands to C-64 BASIC. As with Vic Tree, the main additions are for disk handling and editing. For example, disk commands are provided to read the disk directory, and to initialize, rename, copy, or delete files from a disk. Editing commands enable convenient display of program listings, trace, variable search and replacement, renumbering, deleting a range of lines, and program merging. The second Richvale product is SUPERBASIC, which has additional commands for disk maintenance, file maintenance, data handling, string handling, garbage collection, graphics, and error-trapping. A machine language monitor is also included.

Simon's BASIC has recently been announced but is in very short supply. There has been considerable interest in this product because of its many features and low price, but few people have actual experience with it. Commodore's announcement states that Simon's BASIC adds 114 programming commands to the C-64's language. The additional commands enhance string handling, input, screen control, graphics and music generation.

On-Disk BASIC/DOS Extensions

GrafDOS™ is an enhanced DOS that is distributed by Interesting Software, 21101 South Harvard Blvd., Torrance, CA 90501. This DOS provides several additional commands for disk control, screen display, and graphics. In many ways, it makes available on the C-64 a BASIC very similar to the Applesoft™ BASIC used in Apple computers. Many grafDOS statements even use Apple reserved words. For disk handling, additional commands are provided for cataloging the disk, loading and saving binary files, copying files, chaining, and controlling memory. VTAB and HTAB statements have been provided for cursor control, as well as

several low-and high-resolution graphics commands that are Applesoft clones. The grafDOS disk includes a machine language minitor.

Selecting a Utility to Suit You

The above description will give you a general idea of the types of products available for BASIC/DOS enhancement. There are many more products than I had space to describe. Moreover, products change, and new products are constantly being introduced. In short, do not feel limited to just these products. Visit your Commodore dealer, check the magazine ads, and talk to your friends. See what the various products can do, how their features match your needs, and what dealers and other programmers recommend.

The following is a shopping list of commands and program functions that I consider essential:

- Catalog (or directory)—Permits you to display the disk directory without replacing the program in memory.
- Rename—Renames a disk file.
- Scratch—Deletes a disk file.
- Delete (line x—line y)—Deletes all the program lines between two specified line numbers.
- Renumber—Renumbers program lines.
- Merge—Merges two programs into one.
- Global Search and Replace—Searches through your program to find and change the label of a variable or character string you specify to a new label you specify.
- Variable Dump—Displays all variables currently in use in a program.
- Copy—A good, fast, disk-drive copy utility.

Two additional utilities that are extremely useful, although less critical, than those just mentioned are:

- File Compactor—Removes remarks from program and compresses the program further by fitting as many statements onto as few lines as possible. File compression reduces file size by as much as 50 percent, and increases program execution speed dramatically. It is the next best thing to compiling a program.
- Line Cross-Referencer—Tells which lines are called from other lines with GOTO or GOSUB statements.

Your User's Group

There are public domain (that is, noncopyrighted) programs for disk copying and file compression available. User's group libraries often include utilities that perform these and some of the other functions as well.

In addition, a user's group is a good place to get additional information about program development utilities that may suit your needs. If you do not already belong to such a group, join one. Even if you are a recluse, and antisocial by nature, the free programs and consulting you can get make it worth it.

Another source of public domain software is a company with the interesting name of Public Domain, Inc.™, 5025 South Rangeline Road, West Milton, OH 45383. This company is a sort of clearinghouse for public domain software, and collects and copies many different programs for Commodore-series computers. Prices are reasonable. Write them for their free catalog.

BOOKS AND MAGAZINES

The best sources of information about the C-64, programming, and new hardware and software are books and magazines. The serious C-64 programmer probably subscribes to several magazines and has a shelf full of computer books. The publications of interest fall into three general categories:

- Magazines—Provide programs, programming tips, and news about software and hardware developments.
- C-64 books—Provide information on the C-64 that is broader in scope than a magazine article.
- General books—Provide information on topics of interest to the programmer—programming techniques, hardware, writing, and so forth.

This section provides brief descriptions of publications that I have found particularly useful. Some of these are necessary, and others optional. The list is far from complete, but everything on it is useful, and it is a good place for you to start, if you do not subscribe to the magazines, or have the books already.

Books

The books described below are necessary.

Commodore 64 User's Guide (Commodore Business Machines, Inc., and Howard W. Sams & Company, Inc.,). This is the user's guide for your C-64, and came with your computer. It gives a good introduction to the C-64 for the beginner, but the experienced programmer will soon move past it.

VIC-1541 Single Drive Floppy Disk User's Manual (Commodore Computer). This manual comes with the VIC-1541 disk drive. It is Commodore's description of how to use your disk drive. It is badly written, confusing, and contains errors. It is a source document however, and it contains important information that you must know to use your disk drive.

VIC-1525 Graphic Printer User's Manual (Commodore Computer). This is the user's manual for the standard printer used with the C-64. If you purchase a VIC-1525 printer, then this will come to you with it. If you use another type of printer, then this manual is still important, since it tells how to control your printer from within a program. The printer manual is just as readable, enlightening, accurate, and important as Commodore's manual for the disk drive. Fortunately, the printer manual is shorter, and less apt to confuse.

Commodore 64 Programmer's Reference Guide (Commodore Business Machines, Inc., and Howard W. Sams & Company, Inc.). This guide contains nearly 500 pages of useful information. It is well organized, well written, easy to use, amply illustrated, and comprehensive. Get it.

The following book is highly recommended.

Your Commodore 64: A Guide to the Commodore 64 Computer, Heilborn & Talbott (Osborne/McGraw-Hill). This is the best of the introductory, system-specific books on the Commodore 64, and is useful to have on your bookshelf even if you are not a programming novice. It introduces the C-64, and provides a variety of introductory and advanced C-64 programming information.

The following three books have nothing to do with the C-64, but are recommended nonetheless.

The Elements of Style, Strunk & White (Macmillan). This is a book about writing. It is a classic, 71 pages long. It offers 39 rules for effective written communication that you can actually understand and apply. This book has influenced not only writers, but also had a strong impact on programmers. The two classic works on programming technique, described below, were modeled after it.

The Elements of Programming Style, Kernighan & Plauger (McGraw-Hill). Kernighan and Plauger are to programming technique what Gilbert & Sullivan were to the operetta—the guys who set the standard and who everyone copied or acknowledged. This book offers simple rules for writing solid programs and developing a good programming style.

BASIC with Style, Nagin & Ledgard (Haydn). This is a concise book on BASIC programming technique, close in spirit to Strunk & White's book on writing. All programmers in BASIC should read it.

Magazines

The following magazines provide useful information on the C-64.

Compute! This monthly magazine covers many popular microcomputers, but gives a special emphasis to the C-64. The editorial staff is strong and the articles are generally excellent. Strongly recommended.

Compute's Gazette. By the publishers of *Compute!*, this magazine carries articles dealing exclusively with the C-64 and VIC-20. Its slant is toward the beginner, but it provides useful information for all C-64 users.

Transactor. This is a bi-monthly Canadian magazine that is directed at users of all Commodore computers. It is written for and by Commodore enthusiasts, which may be a little unhealthy, but which guarantees a high standard of technical ex-

cellence. There is little advertising, and the articles read like letters among friends.

Run. This Wayne Green publication deals exclusively with the C-64 and VIC-20 computers. It has good articles and programs.

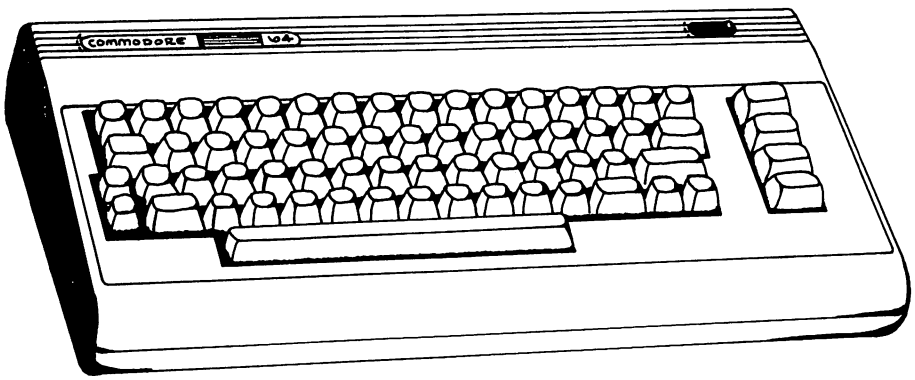
Creative Computing. This magazine, like *Microcomputing,* covers the microcomputer field, but has regular columns and articles on the C-64.

Commodore. This is a bi-monthly magazine on the C-64 that is published by Commodore Computer.

BYTE. It is the best technical magazine covering the entire microcomputer field. If you want to know what is going on beyond the Commodore line, subscribe to it.

InfoWorld. This is a weekly magazine that provides news on the microcomputer industry. It consistently prints the most up-to-date information you can get. Unlike most magazines, which have a three-month delay between creating editorial content and publishing, InfoWorld works on a weekly deadline. Generally entertaining, it is must reading for anyone seriously interested in the industry.

Chapter 3



Programming Tips and System Documentation

This chapter moves beyond the introductory material in Chapters 1 and 2 and gets into actual programming. It contains a series of C-64 programming tips designed to make your life as a programmer easier. Most of the information offered is based on my experience as a programmer, or the experience of other C-64 programmers whom I know.

If you have programmed the C-64, you realize that it has some quirks. Hence, this chapter gives you a few warnings—a sort of map of the C-64's minefield. It also shows you a good way to structure a typical program. It describes the content and organization of the model program that is used throughout the rest of the book. This chapter presents the program framework; later chapters fill in the details.

The second half of the chapter covers the subject of system documentation. This is documentation written by programmers for other programmers. It is used to communicate how a program works in sufficient detail that its target audience can make sense of the mysterious code comprising the program. System documentation is generally regarded as, well, a dull and uninteresting subject. I cannot liven it up much, unfortunately, but I will at least attempt to keep it from becoming tedious. As a subject, system documentation is not nearly as interesting as programming, but it is of nearly equal importance. Horrendous problems occur when programs are not documented properly, as you well know, but more about that later.

Regard what is presented in this chapter as a set of recommendations, rather than a prescription that you must follow rigidly. How you design a program is very much a matter of style and personal preference. No one can write a rule book that works for everyone. In defense of what is presented here, these techniques have worked well for me, and for others whom I know. They were not developed sitting behind a desk, but through a good deal of struggle and experience. They should help you as well. By following them, you may be spared some of those unpleasant occurrences that are afterward referred to philosophically as "learning experiences."

I assume that, as you begin this chapter, you

have read Chapter 2, done some research, and are not working with a "naked" C-64. You should have a disk drive and printer attached to your C-64, upgraded your BASIC/DOS with either a plug-in ROM or a disk-based enhancement, and have or be acquiring the utility programs recommended in Chapter 2. You will need this hardware and software from now on.

PROGRAMMING TIPS

This section presents eight programming tips, including obvious things like backing up your files, and more complex things like making your program readable. They were not handed to me on stone tablets, nor are they numbered with Roman numerals. Still, they are commandments of a sort.

Back Up Your Files

Ever lost a program or data file? If not, you are the first programmer in history. It happens to everyone, even experienced programmers.

The disk may become physically damaged. You know, you get frustrated late at night and attempt to jam the disk into the drive, or some other physical misfortune befalls it. The disk may wear out. This is rare, but it can occur.

The most likely problem, however, is that some sort of human error occurs. You may overwrite a file. The file may, through quirks in the C-64's operating system, pick up some bizarre refuse that does not belong in it. In a moment of carelessness, you may leave the disk lying on top of your 1954 Philco TV set, where the yoke of the picture tube scrambles the disk's magnetic medium.

The only solution to such calamities is to back up your disks, and back them up with a sort of single-minded compulsiveness. You must back up any disk that is written to regularly. This includes both program and data disks, and also to any utilities disks that are written to.

This is why you need a good disk-disk copy utility. It is necessary for copying data disks, and handy for copying program disks. The copy utility is usually used to back up data disks on some systematic basis, such as once a week or after each update is made to a set of data files. Backing up files this way is straightforward: decide how often the disk needs to be backed up, and then get in the habit of doing it.

Backing up program disks is different from backing up data disks because, for a programmer at least, a program disk is in a constant state of flux. During program development, the program constantly changes as newer versions replace older versions. What is the best way to handle backups in this situation?

One common way is to use two disks, labeled, say, "Disk 1" and "Disk 2." The programmer works mainly with Disk 1, saving the developing program to it frequently. At intervals, Disk 1 is copied to Disk 2. How often? This usually depends on the programmer's paranoia and the number of recent bad experiences in losing program files. Basically, a trade-off is involved among the probability of Disk 1 becoming bad, the probability that Disk 1 already is bad, and the work involved in backing up Disk 1.

Another way of looking at it is that you do disk backup for insurance. Disk 2 is the insurance policy. If what you copy to it from Disk 1 is bad, however, you just lost your policy. There is always the chance that Disk 1 may go bad in some way that you do not detect. For this reason, I suggest a somewhat more cautious approach to doing disk backups. It involves three disks instead of two. The recommended procedure is illustrated in Fig. 3-1, and works like this:

- Use three disks labeled "Disk 1," "Disk 2," and "Disk 3."
- Make Disk 1 your master. Work with it and keep the most current versions of all programs on it.
- As you make program changes, periodically switch between Disk 1 and Disk 2. Finish your working session with Disk 1 in the drive.
- At the end of the session, transfer your work to Disk 3. Disk 3 is your "archive" copy. It is what you fall back on if you lose useful copies of your work on both Disk 1 and Disk 2 during a working session. Do

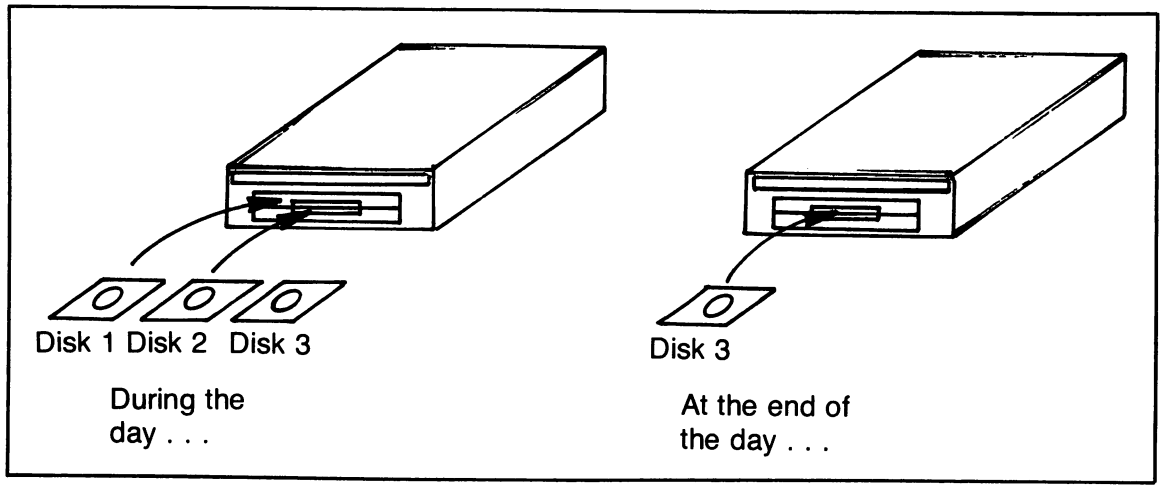


Fig.3-1. A three-disk backup procedure provides extra insurance against the loss of program files during program development. Disks 1 and 2 are periodically switched throughout the working day, and Disk 3 is copied only at the end of the session. Disk 3 thereby provides backup in case both Disks 1 and 2 lose an important file.

not transfer information to Disk 3 with a disk-disk copy program. Save your work with the SAVE command (use the SAVE technique described under the next tip).

Disk 3 is the key to this copying technique. You update it once per working session, it contains draft versions of the program, and it is not meant to be a perfect copy of the master. Rather, it contains snapshots of the program across time. If something should go wrong with both Disk 1 and Disk 2, you can pull Disk 3 out of your file and use it for a fresh start.

Never transfer all of Disk 1 to Disk 2 until you are sure that every file on Disk 1 is sound. The only way you can test this is to RUN the programs that Disk 1 contains or LOAD each program separately and go through its listings to check it.

Save Programs Carefully (Beware the Bug!)

As you develop a program, you must SAVE it to disk countless times. With the C-64, this is a particular nuisance. The syntax of the SAVE command is cumbersome when you SAVE a program to a disk that already contains a file with the name of the program you are saving.

Here is a little tip that will reduce the effort involved. Make line 1 of your program a REMark statement that looks like this:

```
1 REM SAVE" @0:PROGRAM NAME",8
```

Substitute your program's name for PROGRAM NAME. To SAVE the program, LIST line 1, space over the line number and the REM statement, and press the Return key. Your program will be SAVEd. This is much easier than remembering the syntax and program name, and then typing the entire SAVE command in from scratch each time. It is also less likely to cause an error.

There is nothing magic about using line 1, of course. You can use any line (except line 0, which cannot be listed), but I suggest 1 because it is short and at the beginning of your program where it can be used for program identification. If you work on only one program at a time, identification is not a problem. If you work on several, and parts of them are interchangeable, you must identify each program somewhere. The SAVE REM is a good place to do it. Now that you know an easy way to SAVE programs, let us see why saving is more difficult than as just described.

Here is something that occasionally happens to C-64 programmers. They LOAD up a program, make some change to it, then save it on disk. Then, they LOAD a second program, and SAVE it. They may do this several times. Later on, they attempt to

LOAD one of these programs and discover that the program in memory is not the one whose name they typed in with the LOAD command.

Has this ever happened to you? If so, you may have had a sinking feeling, panic, or seen your life flash before you. You may have concluded that you SAVEd the program in memory with the wrong name. Human error, in other words—it is easy to blame this phenomenon on carelessness, stupidity, working too late, drinking too much coffee, or the glass of wine with dinner. While such factors sometimes are the cause, the C-64 has an alarming tendency to make this error itself. Even if you put the SAVE command in a REMark line—which ensures that the program will be SAVEd with its correct name—this problem sometimes occurs. I said sometimes. It is unpredictable. It is one of the dangerous aspects of programming your C-64. It is dangerous because you can lose a file this way, just as if it passed into the fourth dimension. If you cannot LOAD it into memory, it is gone.

Why does the C-64 do this? The answer depends on whom you ask. The problem is not well documented, and so there is no scientifically sound explanation. One theory is that there is a bug in the operating system that awaits the right, mysterious set of conditions to act. If you accept this theory, you might think of your C-64 as haunted, or subject to the whims of a poltergeist. Another theory is that the C-64's DOS sometimes gets tired, loses it directory, and may then link two different names with the same physical file. There are probably other theories for the phenomenon that I have not yet discovered.

All of which is interesting, but not very helpful, since none of the theories is sufficiently rich or robust to enable prediction of the C-64's aberrant behavior. In short, we have a problem, folks, but do not really know what causes it.

The only solution is to work with one program at a time, and to restart your computer each time you begin working with another program. You can restart your computer by turning it off and then on again. An easier way to restart is to type in SYS 64738 and press the Return key. This clears memory and performs a warm start on the C-64. Doing this repeatedly may cause the C-64 to hang up. If this happens, turn it off and then back on again.

As long as you work on one program at a time, you should have no problem with file mix-ups. If you do work on more than one, follow this procedure to avoid any possibility of losing your valuable program files.

You will not find that in your C-64 documentation!

Plan Your Variable Assignments

Many programmers assign variables as they develop a program, sort of pulling them out of the air. Some new part of the program must be written, and the programmer realizes that this requires a variable, an array, or whatever. The programmer then asks him or herself what variable to use. The assignment cannot be made carelessly, or confusion with other variables may occur. The programmer then tries to recall what variables are already in use, and make up one that is not. In large programs this is difficult, as there may be scores or even hundreds of variables. One solution is to pick some really improbable variable name and use that. The alternative is to look at the program's listing and check every variable in use. Sound familiar?

We all do this, and can get away with it to a certain extent. If the program is small, then we really can keep track of all the variables (and functions and arrays) inside of our heads. Beyond a certain point, however, we are only kidding ourselves, and risking some nasty variable confusions.

The best way to prevent variable confusions is to be systematic about making and keeping track of variable assignments. Make as many assignments as you can before you start coding. Decide what local and global variables you will need and assign as many of them as you can at the beginning. *Local variables* are the work horses in most programs—they are used often, on a temporary basis, in many parts of a program. They are the faceless ones whose identities are constantly changing. *Global variables* have greater integrity—they are used to represent one thing.

After you have decided what variables you need, make a record that lists each one. For global

variables, list each variable and what it represents. Local variables usually have many different identities, often very briefly, and so it does not always make sense to tell what they stand for. Make a list of them, however. Examples of such documentation appear in the System Documentation section of this chapter.

You cannot possibly anticipate all variable assignments at the start—so you must assign some new variables later on. When the time comes, check your records to see what variables are in use. By having a record, you can prevent making duplicate assignments. After you make the new assignment, document it. In short, keep a careful record as you go.

What should you name variables? The current thinking among people who study program documentation is that the more descriptive the label, the more readable the program. For example, if you want to store the value of a Sale Price in a variable, it would be a good idea to use something like SALEPRICE. Likewise, a good string variable for storing someone's Name would be NAME$. It does not make much sense to follow this rule with local variables, since they do not represent a single thing, but it does make sense for global variables.

There are some problems with using long variable names. One is that, if you do it carelessly, you may produce variable confusions. Although C-64 BASIC allows you to represent a variable with up to 12 characters, it only pays attention to the first two characters. Thus, it does not discriminate, for example, between the two variables FARE and FACT. If you use long variable names, be very careful to avoid such confusions.

A second problem with long names is that they take up more memory. How much more depends upon the number of variables, but in most cases it will not have a serious overhead cost.

Variable labels do not have to be complete words to be informative. An abbreviation that has a mnemonic relationship to the thing being represented is better than a label that has no relationship at all. For example, if you do not want to use SALEPRICE and NAME$, S and N$ are better than X1 and Q$. S and N$ bear a mnemonic relationship to Sale Price and Name, but X1 and Q$ do not. The labels that you assign—long, short, informative, uninformative—are much a matter of style and personal preference.

Within this book, I tend not to use long variable labels, since this is my programming style, although I do attempt to make the global variables meaningful mnemonics of what they represent. In some programs, long labels may be better than the short ones that I use, but I have found this style very workable in a wide variety of programs.

No one keeps perfect records, of course; you may reach a point during coding where your records say that a particular variable is not in use, but you are not sure whether or not to believe them. At such times, it is very helpful to have a utility that will create a list of program variables. Some BASIC/DOS enhancements, such as Vic Tree, permit you to dump all of the variables referenced in a program to your screen or printer. You can then check your record. You can do much the same thing by using a global search and replacement utility. When it comes time to make the assignment, it may occur to you that a particular variable would be logical on the basis of its mnenomic qualities. Use your global search and replacement utility to search your program to see if that variable is already in use. (Do not make any replacements, of course, or simply replace the variable with itself.) This will tell you if you can make the new assignment without conflicts.

Make Your Program Readable

The BASIC programming language is widely criticized by computer scientists and others for its lack of readability. Some feel that the language itself, with its flexible control structure, ability to define data structures in the course of a program, and mischievous GOTO statement, have cursed it with a sort of Original Sin that renders any program written in it unreadable.

The problem is not, however entirely in the BASIC language. Much of it lies in bad examples and bad advice about how to use it. The BASIC program examples printed in popular magazines tend to support the academicians' contentions, usu-

ally being incomprehensible without a written explanation. You can make a BASIC program very readable if you organize it logically, use REMarks, and put only one program statement on each line. How many programs have you seen that were coded this way? Probably very few.

To add to the confusion, many authoritative sources offer programmers advice that, if followed, will tend to make their programs less readable. For example, the C-64 Programmers's Reference Guide states on page 25 that you can increase the speed and reduce the memory required by a program by leaving out REMarks and putting as many statements of code on each line as possible. This information is accurate, of course, but given without qualifications. Program speed is a good thing, and so is making a minimum demand on computer memory. Leaving out REMarks and bunching up program statements, however, makes it much more difficult to read a program and understand what is going on in it.

Actually, you can have it both ways—have a program that is both readable and fast—by creating two different versions of it. Code the program first for readability, and then "optimize" it with a file compression utility. A file compression utility of the sort described in Chapter 2 will remove REMarks, and compress program statements onto as few lines as possible. Utilities are much better at doing this sort of thing than people. They do not mind working with long lines of code, and are indifferent to REMarks. A programmer, on the other hand, finds it much easier to figure out that he or she is working on the Analysis Display Generator if there is a line of code containing a REMark that says ANALYSIS DISPLAY GENERATOR. Likewise, it is easier to change code if each statement is on a separate line.

Many of us have gotten the idea that using REMarks is wasteful, a sign of amateurishness, or something we should avoid. We may have similar ideas about using only one program statement per line. These ideas, if we have them, are wrong. Forget what some programming manuals say, what bad code examples you have seen in magazines or books, or what your favorite computer genius does.

Instead, write your program for clarity. Write it so that others will understand it. In doing this, you will make it much easier to understand yourself, make it easier to modify, and reduce the chances of errors. Here are two simple rules for making a program readable:

- Use REMarks liberally—Use them to identify the program modules and submodules within your program, to tell what is going on. Use enough REMarks to tell the story.
- Use one program statement per line—This will make your program easier to read and to modify. After you see code written this way for a while, you begin to like it. There is a sort of elegant simplicity in this style. It suggests an ordered mind, good work habits, solid character, and respect for one's fellow human beings. (Feel free to add other virtues to the list.)

These simple rules can make a world of difference in the readability of your programs. To illustrate, compare Figs. 3-2 and 3-3. The program listing in Fig. 3-2 follows these rules. The listing in Fig. 3-3 does not. Decide which one you would find the easiest to work with.

A few additional points about file compression follow.

When you compress files with a utility, save an uncompressed source code version of your program. You will need the source code version for making program changes later. A file compression utility will often pack the lines so tightly that insertions are impossible. Thus, to make changes, you must go back to the source code version, modify it, and then recompress it. Maintain source code versions of programs on different disks from compressed versions. The source code is important. Treat it with great care. I suggest putting it safely away in a file with the program listing, rather than keeping it out among diskettes you actively use.

Always make a printed listing of the source code version of your program. Making a listing of the compressed version is optional. It may help you troubleshoot a problem in the program, but it is fairly easy to relate a compressed program to the

```
12000 REM--GET DATA DISK ID--
12010 INPUT"TYPE IN DATA DISK I.D.: ";ID$
12020 IF LEN(ID$)=0 OR LEN(ID$)>12 GOTO 12010:REM ID LENGTH TEST
12030 REM--DIM NEW ARRAYS--
12040 A=A%(0)
12050 DIM O$(A),O(A),O1(A),O1%(A),O2%(A),O3%(A),O4%(A)
12060 A=A%(1)
12070 DIM P$(A),P(A),P%(A)
12080 REM--INITIALIZE FILES--
12090 REM-CLEAR DATA FILES OF OLD VALUES-
12100 REM-BUDGET CATEGORY #1-
12110 FORA=0 TO A%(0)
12120 O$(A)="-":O(A)=0:O1%(A)=0:O2%(A)=0:O3%(A)=0:O4%(A)=0
12130 NEXT
12140 REM-BUDGET CATEGORY #2-
12150 FOR A=0 TO A%(1)
12160 P$(A)="-":P(A)=0:P%(A)=0
12170 NEXT
12180 C1$="UP":REM SET WRITE FLAG
12190 REM-WRITE FILES-
12200 GOSUB 7000:REM WRITE DATA FILE
12210 GOSUB 7500:REM WRITE MARK FILE
12220 GOTO 10000:REM RETURN TO MAIN MENU
```

Fig. 3-2. Listing of a portion of a program with sufficient remarks to make it readable.

```
12010 INPUT"TYPE IN DATA DISK I.D.: ";ID$
12020 IF LEN(ID$)=0 OR LEN(ID$)>12 GOTO 12010
12040 A=A%(0)
12050 DIM O$(A),O(A),O1(A),O1%(A),O2%(A),O3%(A),O4%(A)
12060 A=A%(1)
12070 DIM P$(A),P(A),P%(A)
12110 FORA=0 TO A%(0)
12120 O$(A)="-":O(A)=0:O1%(A)=0:O2%(A)=0:O3%(A)=0:O4%(A)=0
12130 NEXT
12150 FOR A=0 TO A%(1)
12160 P$(A)="-":P(A)=0:P%(A)=0
12170 NEXT
12180 C1$="UP"
12200 GOSUB 7000
12210 GOSUB 7500
12220 GOTO 10000
```

Fig. 3-3. Listing of the same portion of the program shown in Fig. 3-2, but without any remarks—although the BASIC statements are recognizable, the purpose of this program and what is going on in it are impossible to determine based on the listing alone.

source code listing by following GOTO and GOSUB statements and variable names. If some problem does exist in the compressed version, attempt to produce the same problem in the source code version, and then work with it to correct the problem.

Generally, it is not advisable to attempt to make changes to a compressed program because the statements are packed too closely. In fact, in many cases, the compression utility puts more statements on a single line than could be typed in from the keyboard. It can do this by using statement tokens that it stores internally and can display, but whose BASIC statement equivalents take up more space than can be displayed on a maximum-length (160 characters) C-64 program line.

Plan and Organize Your Programs Consistently

Among creative people—and programmers usually are creative—consistency is not always a popular idea. The creative person does not like to be hemmed in by rules that hinder freedom of action. There is, however, a powerful argument for consistency in programming: it reduces the amount that you must learn and remember, and thereby reduces the likelihood of errors. If you do things consistently, then there is less to learn, less to remember, and a smaller chance of misrecalling something and making a mistake.

If every one of your programs looks different from every other one, then the mysteries of each one must be fathomed separately. Each new program is a new adventure, planned from scratch, executed, and then put into use. Later on, when you go back to it, you must figure it out all over again.

A more sensible approach is to build all programs on the same basic model. Obviously, there is a limit to how completely you can do this. Different programs will do different things and require different building blocks. A surprising amount of what you do from program to program remains the same, however. Usually a program requires you to do some or all of the following:

- Generate displays.
- Take user inputs.
- Control your program.
- Read and write files.
- Operate your printer.

Often, you can use the same (or very similar) subroutines for doing these things in many different programs. If you do this, then your programs begin to look similar, and this makes each one easier to understand and work on. Although such consistency is obviously not for everyone or for every program, it is a good ideal to keep in mind. If your programming style and application permit, you may find that it works very well for you. If you have not been programming this way in the past, you may need to get used to it.

To get started in this direction, you must first devise a plan (or *architecture*, to use the fancy word) that you can use as the framework for designing many of your programs. As you develop programs, start with this framework. Modify it to suit the specific application. If you do this, you will find that developing a new program becomes less a matter of creating spun cloth from thin air and more a matter of taking parts of what you have done before, adapting them, and adding the new pieces that are necessary. All programmers do this to some degree, but some are more systematic about it than others. I suggest that you try to be very systematic, if your personality and programming style permit it.

To illustrate one way of building a program framework, I will describe a model program. Different parts of this program contain different things. Later in this book, I will fill in the details by showing subroutines and other elements of code that fit in each part of the model. The model is illustrated in Fig. 3-4, and is divided into three parts:

- Setup Routines (lines 0-999).
- Subroutines (lines 1000-9999).
- Main part of program (lines 10000-63999).

When the program first starts, the setup routines are executed. They prepare the display, dimension arrays, define functions, and so forth.

The last line in the Setup Routines is the statement GOTO 10000, which jumps over the subroutines to the main part of the program. Subroutines are placed early in the program to increase their speed of execution. The lower their line numbers, the more quickly the program can access and execute them.

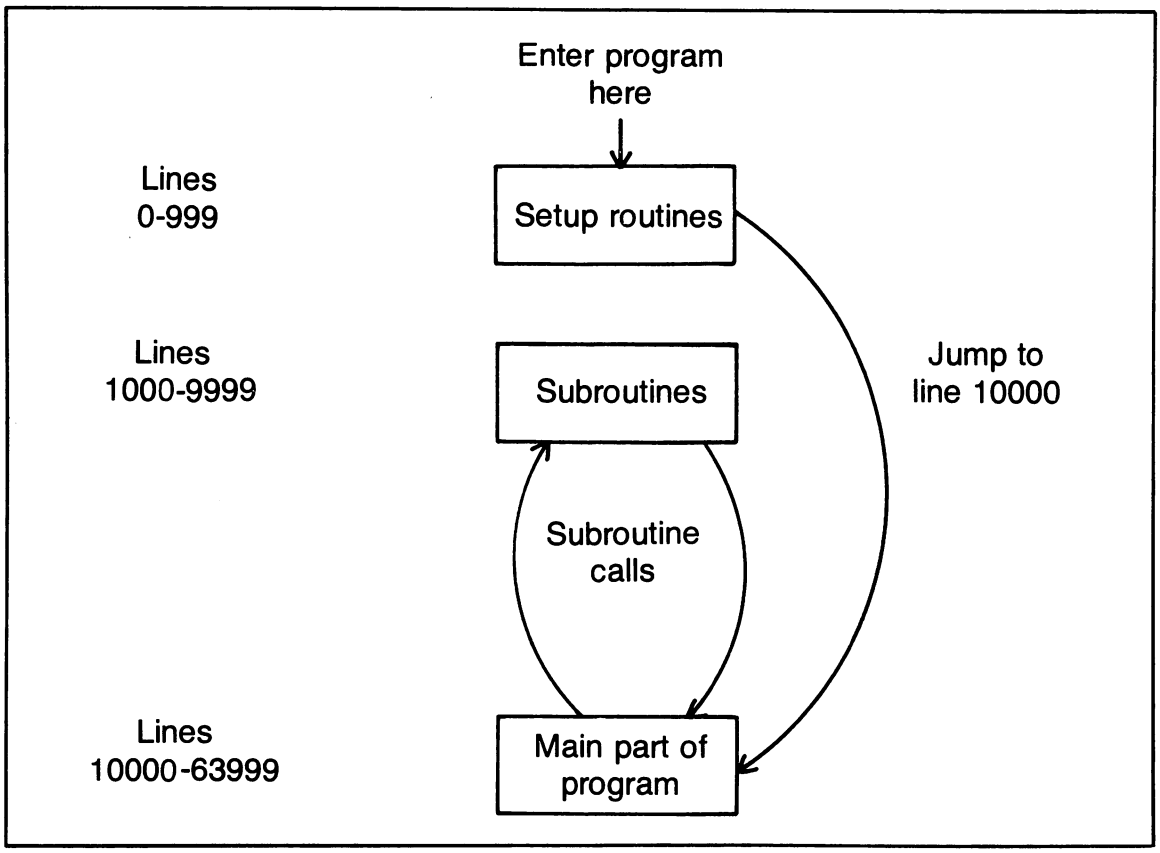


Fig. 3-4. A common way to build a program is to break it into three main parts, as shown here. Setup routines are placed first, followed by Subroutines, which are given low line numbers to increase their speed of access. The Main Part of the program begins at line 10000.

The real action in the program is in its Main Part. The program's control structure and subroutine calls originate from there, and this is where the program's actual intelligence lies. The Setup Routines lay the program's foundation, and the Subroutines are workers called on to do different jobs. What makes the program unique, however, is what lies in its Main Part. In fact, most of what is transferrable from one program to another is what exists in the Setup Routines and Subroutines.

Most experienced programmers place frequently accessed subroutines early in their program. Some programmers prefer to place them ahead of everything else, and use the program organization as shown in Fig. 3-5. The difference between this organization and that shown in Fig. 3-4 is that the locations of Setup Routines and Subroutines are reversed. This organization is satisfactory, but I prefer that shown in Fig. 3-4, and I will use it as the model in this book.

Returning to Fig. 3-4, let us consider what goes on inside each of the three main blocks of the program.

Setup Routines (Lines 0-999). When a C-64 program begins to execute, it must usually perform several functions to initialize the program. These functions generally include some or all of the following:

- Clear display screen.
- Display the program's title page.
- Disconnect the RUN/STOP and RESTORE keys to prevent the program from

interrupting if one of these keys is touched by mistake during the program.

- Set the colors of the display border, background, and characters.
- Set uppercase or lowercase display.
- Assign constants with assignment statements and by READing DATA statements.
- Dimension arrays.
- Define functions used in the program.
- Obtain required inputs from the operator.
- Read files necessary to initialize the program.

Most of these functions are optional, and some programs require additional functions to be performed during initialization. These functions are, however, typical of many programs. The order in which they are performed is arbitrary, although some things must be done before others. In general, it is best to clear the display screen early, set the colors, and present the title page, if one is used—this keeps the user occupied, and averts the impression that something is wrong as the computer sits there with a blank screen. RUN/STOP and RESTORE keys should be disabled before taking operator inputs. Arrays must be dimensioned before assigning values to them with READ statements or reading files that use the arrays.

Figure 3-6 contains the listing of the setup routines in a typical program. This code performs all of the functions described earlier, and also includes two REMark lines that can be used to SAVE the program (line 1) and print its listing (line 2), a copyright statement (lines 10-20), and a date (line 30).

Line 0 contains a REMark statement with the program's title.

Line 1 contains the REMark statement for SAVEing the program.

Line 2 contains a REMark statement for printing the listing of the program (see the Proceduralize Step).

Line 5 clears the screen.

Lines 10 and 20 contain a copyright statement.

Line 30 is the date the program was last up-

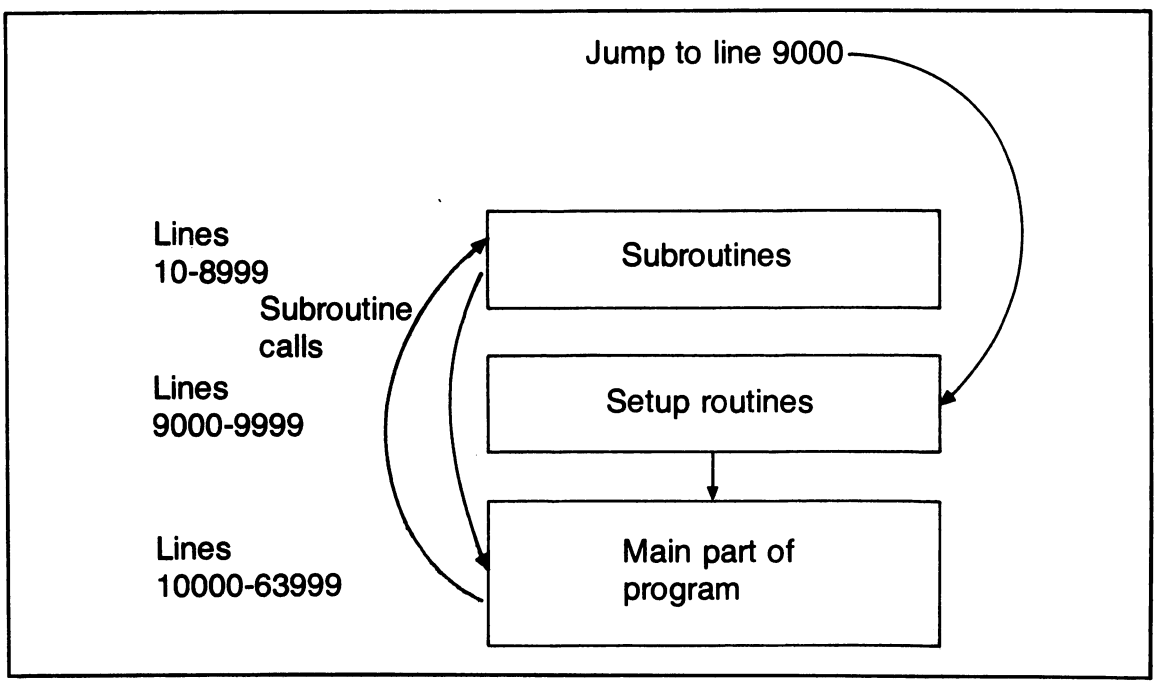


Fig. 3-5. An alternative to the program organization shown in Fig. 3-4. Here, the subroutines are placed at the very beginning of the program, and setup routines afterward. This organization may result in faster access to frequently used subroutines.

```
0 REM SETUP ROUTINES
1 REM SAVE"@0:FIGURE 3-6",8
2 REM OPEN1,4:CMD1:LIST
5 PRINT CHR$(147):REM CLEAR SCREEN
10 REM COPYRIGHT (C) 1984 BY HENRY SIMPSON
20 REM ALL RIGHTS RESERVED
30 REM 6/15/84
40 POKE 808,225:REM DISABLE RUN/STOP & RESTORE KEYS
100 REM--SET UP DISPLAY--
110 POKE 53280,0:REM SET BORDER COLOR=BLACK
120 POKE 53281,0:REM SET BACKGROUND COLOR=BLACK
130 PRINT CHR$(158):REM SET CHARACTER COLOR=YELLOW
140 PRINT CHR$(142):REM SET UPPERCASE DISPLAY
200 REM--ASSIGN CONSTANTS--
210 C1=10
220 C5=.5
300 REM--DIMENSION ARRAYS--
310 DIM MO$(12):REM MONTH OF YEAR
400 REM--DEFINE FUNCTIONS--
410 DEF FN C1(X)=INT(X*C1+C5)/C1:REM ROUND TO ONE DECIMAL PLACE
500 REM--READ MONTHS OF YEAR--
510 DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
520 FOR A=1 TO 12
530 READ MO$(A)
540 NEXT
600 REM--OPERATOR INPUT--
610 INPUT"PLEASE TYPE IN YOUR NAME: ";NAME$
620 IF NAME$="" GOTO 610
700 GOSUB 6200:REM READ SETUP FILE
990 GOTO 10000:REM JUMP TO MAIN PART OF PROGRAM
```

Fig. 3-6. Setup routines typical of those used early in a program to initialize the display disable the RUN/STOP and RESTORE keys, dimension arrays, and perform other program startup activities.

dated. The programmer should change this date each time the program is SAVEd.

Line 40 POKEs a value to disable the RUN/STOP and RESTORE keys (see the section that discusses these keys).

Lines 100-130 set the screen's border, background, and character colors (see Chapter 4).

Line 140 sets uppercase display (see Chapter 4).

Lines 200-220 assign constants.

Lines 300-310 dimension arrays.

Lines 400-410 define functions.

Lines 500-540 assign constants by READing DATA statements.

Lines 600-620 collect operator input.

Line 700 calls a subroutine that reads the program's setup file.

Line 990 contains a GOTO statement that jumps program control to line 10000.

Subroutines (Lines 1000-9999). In locating your program's subroutines, you must take three main factors into account. First, you must divide the subroutine line range into different categories, depending on the types of subroutines your program uses. My model uses five categories (Output, Input, Control, Files, Printer) and a catch-all category of Miscellaneous. Second, you must allot enough lines to each category to include all of the subroutines it needs. Third, give the lowest line numbers to the subroutines that will be called most frequently.

The subroutine categories and line ranges assigned in my model program are as follows:

- Output (lines 1000-2999).

- Input (3000-3999).
- Control (4000-5999).
- Files (6000-8499).
- Printer (8500-8999).
- Miscellaneous (9000-9999).

The subroutines presented later in this book are allocated line numbers according to this plan.

Main Part of Program (Lines 10000-63999). This part of the program comes at the end, and occupies the greatest range of lines. It is impossible to create a general model of its content, for it will differ among programs. It will call on subroutines to perform their functions, and use built-in rules or operator input to control the flow of events. Beyond these simple things, little more can be said.

The way this model program is organized is but one of many. The important thing in this model is not so much what line numbers are assigned to different parts of the program, or even what parts of the program are included. Rather, it is that the program has a plan that is easy to describe and understand. The rest of this book will fill in the details of the plan, mainly by creating a set of model subroutines to go in the subroutine section of the model program.

Create a Subroutine Library

If you tend to write programs that perform similar functions, subroutines can simplify your life a lot. Once you develop a good subroutine for reading a sequential file, clearing a portion of your display screen, or generating a menu, the problem is more or less solved. The next time you must perform that function, you can go to that subroutine, use it directly or modify it, and your problem is solved. Not only does this speed things up, but it reduces errors. Perfect and debug a subroutine once, and you never need to do it again.

Proceduralize What You Do

One important reason to use subroutines is that they reduce the amount of thinking that you have to do. You take some complex function, package it in the form of a subroutine, and from that point on, all you need to perform the function is set the arguments and call the subroutine with a GOSUB statement. What is inside the subroutine is transparent, and not really important to you as you write your program. By writing and then using that subroutine, you have proceduralized a complex function.

This use of subroutines is a specific example of the more general principle of proceduralization. The basic idea is that you should try to find ways to package the various functions that must be performed during programming to make them simpler. This applies to other areas of program development as well. Earlier in this chapter I presented this simple, one-line routine for SAVEing a program disk:

```
1 REM SAVE "@0:PROGRAM NAME", 8
```

This is an example of proceduralization. This routine saves time, reduces work, and reduces the demand on memory and the likelihood of error.

While we are on the subject, let us consider two additional routines that come in handy during programming. The first is for generating a hard-copy listing of a program, the second for initializing a disk.

Line 2 of Fig. 3-6 looks like this:

```
2 REM OPEN 1,4:CMD1:LIST
```

You get the idea. This routine works like the one for SAVEing a program to disk. To use either one, LIST the line, space over the line number and REM, and press the Return key. When you do this with line 2, a hard-copy listing of your program will print out. After you generate the listing, deactivate your printer from the keyboard by typing in "PRINT#1", pressing the Return key, and then typing in "CLOSE 1", and pressing the Return key.

Figure 3-7 is the listing of a little program that initializes a disk. LOAD it into memory and RUN it, and it initializes the disk in drive 1.

There are two basic ideas involved in creating and using routines like these. The first is that they save unnecessary work. The second is that they reduce errors.

Reduce work and errors. Proceduralize.

Beware the RUN/STOP and RESTORE Keys

The RUN/STOP and RESTORE keys come in

```
10 OPEN 15,8,15
20 PRINT#15,"NEW0:DISK LABEL,1"
30 CLOSE 15
```

Fig. 3-7. Disk initialization program. Writing a simple program saves the programmer work and reduces errors.

handy when you want to stop a program, but they pose a hazard to program users. It is very easy to hit either one or even both of them accidentally during a program, thereby causing an unintended end to the program. Users often become upset when this happens, especially if they have just spent the last 45 minutes or so painstakingly typing in data or composing their version of the great American short story. In any program that may pose such dangers, it is a good idea to deactivate these two keys during the program, and then to reactivate them when it ends.

You can deactivate the RUN/STOP and RESTORE keys with this statement:

POKE 808,225

(This statement is included as line 40 in Fig. 3-6.) After this statement has been executed, the user can press either of these keys, or both together, and the C-64 remains unimpressed. It is as deaf to these keys then as if the user tapped on the tabletop or scratched a big toe.

You can reactivate both keys with this statement:

POKE 808,237

Make sure that you do reactivate these keys at the end of the program, or else the user may be forced to turn the computer off and then on again to regain control of it. In fact, it is a good policy to return the C-64 to its native state at the end of a program by doing a complete warm start—this clears memory and turns the display to its usual blue color. You can do a warm start with this statement:

SYS 64738

Incidentally, during program development, do not execute a line with POKE 808,225, or you will be unable to stop a program when you want to. The easiest way to have it both in and out is to include it in a REMark line such as this:

40 REM POKE 808,225

After you have developed the program, delete the REM from the line and it will perform as intended.

This is the last programming tip. Now comes the hard part—applying these tips as you program. Some of them will be easier to apply than others. Some of them may fit your style, others not. It is my honest conviction that all of them can help you be a better programmer. It is up to you to put them to work for you.

SYSTEM DOCUMENTATION

Most programmers regard system documentation as a necessary evil and about as exciting as a telephone book, the Dewey decimal system, or the table of organization of large bureaucracy. This is a misperception, however—at least in the eyes of one young novice programmer who took enough interest in it to seek The Ultimate Truth on the subject. This programmer (his name was Fred) trekked many miles and underwent many difficult trials in search of answers. Through his efforts, he discovered the existence of The High Guru of Programming, who resides in an Ashram on a mountaintop in the Far East. Fred obtained an audience with the master, and the following is the substance of their conversation, transcribed from the original parchment transcript. Figure numbers have been added to integrate the master's drawings into this chapter, but the content of the dialogue is otherwise unchanged.

Their conversation took place in a great hall. Fred sat on the floor. The master—an elderly man with a long, gray beard, dressed in saffron robes—sat in the lotus position atop a high, golden pedestal. Herewith, their conversation.

Tell me, master—what is system documentation?

A general definition is that it is information that tells how a program works. It is written by programmers for programmers. It consists of information inside a program in the form of REMarks

and of written information that is outside of the program.

Why should I document my program?

For two reasons. First, to communicate to other programmers how your program works. This knowledge will enable them to fix it if something goes wrong, or if they want to modify it.

Second, because you will forget. Without documenting your program, you will forget many of its details, what the variables stand for, program control flow, and how different subroutines work.

When should I document my program?

Before, during, and after you have written it.

The more you do before, the better. When you plan a program, you can document many things: variable assignments, the design of files, record layouts, what flags are used.

Some documentation is best prepared when coding the program. This is the time to put REMarks in code, for example. Also, if you add new functions, arrays, or variables, or change file or record layouts, document them as you code.

After you finish your program, wrap up loose ends. Clean up your documentation. Do what you should have done earlier, but did not have time to do.

Pray, tell, master, how should I use REMarks in my program?

Use them freely, liberally—like the signs that mark the streets, parks, monuments, buildings, tunnels, movie houses, singles' bars, art galleries, public utilities, dumps, and famous citizens' homes in a city. Leave no mysteries. Tell your secrets.

Label your program modules, submodules, sub-submodules, and so forth. Label your subroutines. Describe the action. Tell what is going on inside your program.

How much of my program should be REMarks?

Somewhere between 10 and 30 percent of all program statements should be REMarks. (Note: The master does not equivocate.)

But won't this make my program slower and use up valuable memory?

Yes and no.

Even if you use many REMarks, they do not have a significant effect on program speed. REMark statements are executed quickly. They affect memory in proportion to the number of REMarks you use. If memory and speed are serious problems in your program, compress it by using a file-compression utility. Always, always keep the source code version of the program for changes, however.

What written documentation should I prepare?

That depends on how complex your program is and whether you are preparing the documentation for yourself or others. The more complex the program, the more documentation you need. If you are writing it for other programmers, you need more than if it is for yourself.

Select what you need from the following list:

- List of functions and their definitions (Fig. 3-8)—Tell what each function does and what its arguments stand for.
- Lists of variables (Fig. 3-9)—List each global variable and tell what it stands for. List local variables alphabetically.
- List of arrays and what each stands for (Fig. 3-10)—Do this just like your listing of variables. If you want to simplify things, make one list that includes variables and arrays.
- Subroutine table (Fig. 3-11)—Prepare tables that give the following information for each subroutine: (1) function (what the subroutine does), (2) arguments, (3) line number.
- File descriptions—Provide detailed descriptions of content and variables, strings, and arrays used. (See Chapter 8 for details.)
- Flowcharts—Do not use these for everything. Use them to explain parts of the program where the control flow is complex.

See Figs. 3-8 through 3-11.

PROGRAM FUNCTIONS

FUNCTION	PURPOSE
FN C1(X)	Rounds X to one decimal place
FN C2(X)	Rounds X to two decimal places
FN D(R)	Calculates data file number based on record number R
-	
-	
-	
FN Z(Z)	Locates last byte in final field of record Z

Fig. 3-8. Suggested format for documenting user-defined functions.

DEFINITIONS OF PROGRAM VARIABLES

VARIABLE	PURPOSE
M	Menu name
M$	Menu number
O	Presentation order
T	Trial number (1-8)
I	Target number (1-16)
J	Data argument
T	Time increment

Fig. 3-9. Suggested format for documenting program variables.

DEFINITIONS OF PROGRAM ARRAYS

ARRAY	PURPOSE
M$(4)	Menu options
V%(21)	Row number
H%(21)	Column number
T$(12)	Target names
R%(128)	Minimum number of responses to targets (1-128)
T(8)	Total time per trial

Fig. 3-10. Suggested format for documenting program arrays.

```
                SUBROUTINE DEFINITION
-------------------------------------------------------
Title          : INPUT# DATA ENTRY
Function       : Prints prompt in reverse video and
                 collects keyed entries from user
Line no.       : 3010
Arguments      : V - screen row (1-24)
                 H - screen column (1-40)
                 P$ - prompt

Returns        : User entry as A$
Side effects   : V, H, P$, A$ assignments affected
-------------------------------------------------------
```

Fig. 3-11. Suggested format for documenting program subroutines.

```
10 REM--DUMMY PROGRAM--

20 REM--VARIABLES, & ARRAYS--

30 H%(21):REM COLUMN NUMBER

40 I:REM TARGET NUMBER (1-16)

50 J:REM DATA ARGUMENT

60 M:REM MENU NAME

70 M$:REM MENU NUMBER

80 M$(4):REM MENU OPTIONS

90 O:REM PRESENTATION ORDER

100 R%(128):REM MINIMUM NUMBER OF RESPONSES

110 T:REM TIME INCREMENT

120 T$(12):REM TARGET NAMES

130 V%(21):REM ROW NUMBER
```

Fig. 3-12. "Dummy" program consisting of a list of variables and arrays and REMark lines containing relevant documentation.

Are there any ways to simplify things?

One trick for preparing lists of functions, variables, or arrays is to make up a dummy program. List each item on a separate line, followed by a REMark giving its definition (Fig. 3-12). This is easier to update than paper documentation.

Another trick is to create a dummy program called "subroutines" that contains a consolidated listing of all your subroutines. If your program consists of several subprograms, it is easier to document your subroutines once in your common listing than to document each subprogram separately.

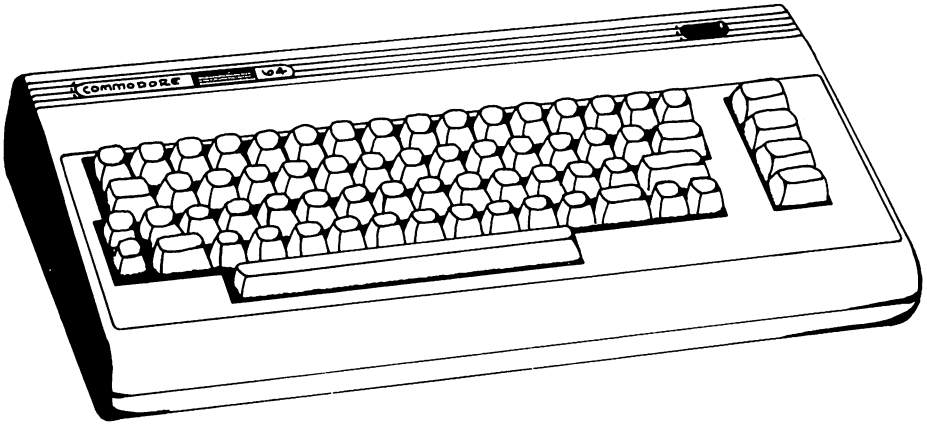
Do you have any other advice to offer, master?

Yes, Be conscientious about documentation. Be compulsive about details. Be consistent. The programmer you save may be yourself.

Thank you, master. You have been very helpful. I have one final question. How can I make my life meaningful?

That is not my department; I suggest you find yourself a good head shrinker. You will, however, be doing yourself and other programmers a service by documenting your programs carefully.

Chapter 4



Output and Screen Design

This chapter covers the fundamentals of program output and display screen design. The computer's display screen—a monitor or television set—is its window onto the world. Through this window the computer presents various types of information to users—English (or other) language statements, numbers, graphics, warnings, requests, directions, and whatever else the clever minds of programmers can invent. The programmer's goal is to design screens that both convey information and are attractive to view. Because both considerations are important, designing display screens is something of an art.

It is also a science, or at least a technology, in that there are a number of practical rules to help you during design. One such rule is to center information on the screen. The idea is simple enough, but it is surprising how many programmers either have not thought about it or simply do not bother. There are many other rules like this.

The main goal of any display screen is to communicate information to the program user. Many factors influence communication:

- The amount of information presented.
- How the information is presented—using numbers, words, or graphics.
- How fast the information comes.
- The colors used.
- Screen layout.

Besides these factors, other less tangible things influence communication, such as how pleasing to the eye the screen appears. Is it possible to design an ugly screen—one so bad that it turns users off? Certainly it is, as many misguided programmers have demonstrated. So, along with providing some of the technical details on screen design, I will work in a few of the aesthetics.

This chapter consists of a series of discussion points. Most of the points are illustrated with examples of C-64 BASIC code. As you read the chapter, and go through the examples, try the techniques out on your own C-64. You will learn more this way, and it will also make the chapter

more interesting. Discussion points covered in this chapter are:

- Design Principles
- Using Color
- Cursor Control
- Clearing a Line or Range of Lines
- Screen Access
- How to Lay Out a Screen
- How to Display Text
- How to Display Numbers

DESIGN PRINCIPLES

The two most important principles for screen design are simplicity and consistency. They are universal principles. They apply not only to screen design, but to how you collect user inputs, control the program, and deal with a host of other design issues.

How do they apply to screen design? Start with simplicity. Keep your screens simple, uncluttered, and focused on a single idea. You will not win any prizes for how much you can fit onto one screen; you will just confuse the person who is using your program. Design the screen so that it has some open space. Present information in bite-sized chunks.

Now take consistency. Display information consistently from screen to screen so it is easier for users to learn and remember how you present information to them. For example, if you display prompts at the bottom of one of your screens, the users will naturally expect them to appear there on other screens. Make it easier for them by fulfilling their expectations. The same idea applies to the other things you present on your screens—titles, menus, warning messages, and the information content of the screen itself. In short, do it the same way from screen to screen.

USING COLOR

The use of color on video displays is complex. It involves human color perception—a difficult subject itself—as well as the characteristics of the hardware and software necessary to generate the displays. The discussion in this section offers practical guidelines for using color and avoiding certain common errors. It focuses on the use of color to convey information in serious programs, not color for flash or appearance.

At least two colors are required to present information—words, graphics, numbers—on any display. Any two colors may be used, but they must differ, and the program user must be able to discriminate between them. One of these colors is the background, or field, against which the information is presented in a second, contrasting color.

When color is used in this way, the main concern is to pick an appropriate color combination for use on the video display. You must select (1) background color, and (2) character color. This section focuses on these two questions.

You can do a lot more with color than I will talk about in this section. For example, use many different colors, and use color-coding so that different colors represent different concepts. Color-coding is, however, beyond the scope of this book. Also, color can be used to add flash and excitement to a program, but these two things do not generally go with serious programs. At any rate, deciding how to use colors this way is more a matter of fine arts than computer programming.

This section begins by describing how to control color on C-64 displays. It then discusses color contrast, selection, and recommended colors for use on displays.

Color Control

C-64 BASIC permits you to set the color of the (1) display field, (2) display frame, and (3) characters.

You set the color of the display field by POKEing in one of the color code values given in the left column of Table 4-1 to memory location 53281. For example, to set the display field to the color black, use the statement POKE 53281,0.

You set the color of the frame by POKEing a color code value into memory location 53280. You set the character color by PRINTing a CHR$ statement using a character code shown in the right column of Table 4-1. For example, to set character

Table 4-1. Background and Character Color Codes.

COLOR	BACKGROUND CODE (Value to POKE)	CHARACTER CODE (CHR$ Value)
Black	0	144
Blue (Dark)	6	31
Blue (Light)	14	154
Brown	9	149
Cyan	3	159
Gray 1	11	151
Gray 2	12	152
Gray 3	15	155
Green (Dark)	5	30
Green (Light)	13	153
Orange	8	129
Purple	4	156
Red (Dark)	2	28
Red (Light)	10	150
White	1	5
Yellow	7	158

color to black, use the statement PRINT CHR$(144).

Putting this all together, if you want to set the display field and frame to black and character color to white, use these three statements in your program:

```
POKE 53281,0
POKE 53280,0
PRINT CHR$(5)
```

In general, there is no good reason for using different display field and frame colors. In other words, when you set colors, POKE the same value into memory locations 53280 and 53281.

As you know, character colors can also be set

```
1020 REM--SET DISPLAY COLORS--
1030 POKE 53280,FRAM:REM SET BORDER COLOR
1040 POKE 53281,SCRN:REM SET SCREEN COLOR
1050 PRINT CHR$(CHR)
1060 RETURN
```

Fig. 4-1. Display color-setting subroutines.

from the keyboard by using the CONTROL or COMMODORE keys in combination with the number keys. For example, you can set character color to black by using a CTRL-1 combination, or to light green using a COMMODORE-6 combination. Program users sometimes do this during a program and the results can be interesting, especially if they set the character color to the background color and then begin to wonder what happened to the display. Well, it is there, but it cannot be seen. To prevent such mischief, it is a good idea to set the display colors at the beginning of the program and then reset them periodically during the program. The most convenient way to do this is to create a subroutine for setting the colors and then simply call this subroutine from time to time. See Fig. 4-1. Subroutine arguments are:

FRAM: frame color code (0-15)
SCRN: screen color code (0-15)
CHR: character color code

To use this subroutine, set the values of FRAM, SCRN, and CHR early in the program and then use a GOSUB 1020 statement to set the color initially and reset it from time to time during the program.

You can also use this subroutine to change colors during the program. This comes in handy under certain circumstances. For example, you might use different color combinations in different parts of your program to help the user remain oriented. You might use a particular color combination to issue a warning, to alert the operator to an error condition, or to attract the user's attention to a situation that requires action.

Obviously, the subroutine is designed to set the background and character colors all at once. If you want to change colors on a character, line, or other basis, then you must insert appropriate CHR$ statements into the code of your program.

In serious programs, be conservative in your use of color. You can do very eye-catching things with it, but that alone does not justify its use. Color for its own sake is fine in games or entertainment programs, but not in serious programs. It can only be justified if it serves a useful purpose. Color is useful as a search aid and as an information code. It can be used for both purposes on C-64 displays. Color is also useful in separating different areas of a display screen; however, because the C-64 creates the entire background color screen at once, it is not easy to draw different colored blocks on the screen that can later be used as backdrops for other information.

Contrast

To display information on a screen, characters must be presented against a background. For example, a screen might contain white characters against a black background. The contrast could be reversed by presenting black characters against a white background.

Which is best—light characters on a dark background, or dark characters on a light background? The answer depends mainly on the surrounding lighting where the screen is viewed. If viewed in a bright area, then it is best to present dark characters on a light background. If viewed in a dark area, then light characters against a dark background are best. The reason is that the eye tends to adapt to the surrounding light level, and if the screen brightness is similar, the eye will see better when viewing the screen.

Generally, high contrast between characters

and background makes displayed information more legible. Using a wide contrast range—such as white on black—is more legible than using a narrow range—such as light blue against dark blue. A wide range can, however, strain the eyes.

It is becoming increasingly common to use dark characters against a light background, although the majority of programs still use light characters against a dark background. Which is best depends upon the situation, as noted earlier. In addition, most users have a preference, one way or another, and these preferences should be taken into account.

Color Selection

Your C-64 can present 16 different background and character colors, and this means that there are 240 different color combinations in which background and character colors differ. That sounds like a lot of possibilities, and it is, in a certain sense. When you start eliminating various colors, however, the numbers drop quickly.

First, consider background colors. Of these, I suggest that you consider only white, the grays, and black. There are three grays—gray 1, gray 2, and gray 3—and placed between white and black, they represent five different shades of gray, or brightness levels. There is no reason to use a color background, other than it may look pretty. The background is used only to contrast with the characters. In addition, some of the colors—such as red, purple, yellow, blue, and orange—conflict with colors that might be used for presenting information. For example, when you put red and blue together, or yellow and green, you get interference at the border that looks like shadows. Okay, so now we are down to five background colors. These colors, and their relative brightness levels, are:

- Black—Dark
- Gray 1—Medium Dark
- Gray 2—Middle Gray
- Gray 3—Medium Bright
- White—Bright

Now consider the character colors. First, the background colors can also be used as character colors, in certain combinations, such as black and white.

Certain colors are seldom used for presenting information. These include blue, brown, cyan, orange, purple, and red. There is nothing wrong with these colors, although the eye is less sensitive to most of them than to the remaining colors. The colors left over are green and yellow.

In short, besides the five colors of the gray scale, the candidate colors for presenting characters are green and yellow. There are two greens, light and dark. The darker one is too dark for use against a dark background, but the light one is perfect. Interestingly, the remaining color combinations approximate those of the three most popular monochrome monitors: white, light green, and amber.

Recommended Colors

Table 4-2 summarizes the recommended background and character color combinations. Four background colors are shown: black, gray 1, gray 3, and white. The recommended character colors for a background color of black and gray 1 are identical, as are those for gray 3 and white. This means that the highest-contrast displays will be created with black or white backgrounds.

Before actually selecting colors for your display, try out several different combinations. Figure 4-2 is the listing of a short program that can be used to generate different background and character colors on your display in order to see how they look. Type the program into your computer and RUN it. When the program starts, you will be asked to enter the background color (a number between 0 and 15 from Table 4-1). The next prompt requests the character color. Type in a CHR$ value from Table 4-1. When you press the Return key, your screen background will turn to the background color selected, and fill with characters of the character color selected. The character color selection prompt will then reappear, enabling you to select and generate new character displays against the background color selected earlier. When you want to select a new background color, interrupt the program with the RUN/STOP key and restart it.

Table 4-2. Recommended Background and Character Color Combinations.

Background Color	Character Colors
Black	White, Yellow, Green (Light), or Gray 3
Gray 1	White, Yellow, Green (Light), or Gray 3
Gray 3	Black, Gray 1, or Gray 2
White	Black, Gray 1, or Gray 2

CURSOR CONTROL

It is important for you to have complete control over the location of the cursor. You must be able to move it where you want it quickly, easily, and with minimum complications. Having this control gives you freedom in how you generate display screens within a program.

It is possible to create a screen without giving much thought to where the cursor is located. For example, you can clear the display, print its top row, then print its next row, and so on until you get down to the bottom (Fig. 4-3). Creating a display this way is not always convenient. It works well enough with simple displays, but not with complex ones.

Often, it is more convenient to print certain parts of the display (such as headings) first, and then to go back later and print other information (Fig. 4-4). To create a screen this way, you must be able to move the cursor to any location very readily. This frees you from the tyranny of line-by-line display generation. New vistas open.

Using Statements

Cursor control with the C-64's 2.0 level BASIC is possible, but cumbersome. To move the cursor to a particular absolute vertical and horizontal position on the screen, three statements are required:

1. Move the cursor to the Home (top left) position by printing CHR$(19) or CHR$ (147). (Printing CHR$(19) is equivalent to pressing the unshifted CLR/Home key.

```
10 INPUT"BACKGROUND: ";A
20 INPUT"CHARACTERS: ";B
30 PRINT CHR$(147)
40 POKE 53280,A:REM SET BORDER COLOR
50 POKE 53281,A:REM SET SCREEN COLOR
60 PRINT CHR$(B)
70 FOR X=1 TO 20
80 PRINT"ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
90 NEXT
100 PRINT
110 PRINT
120 GOTO 20
```

Fig. 4-2. Program for generating display background and character color combinations.

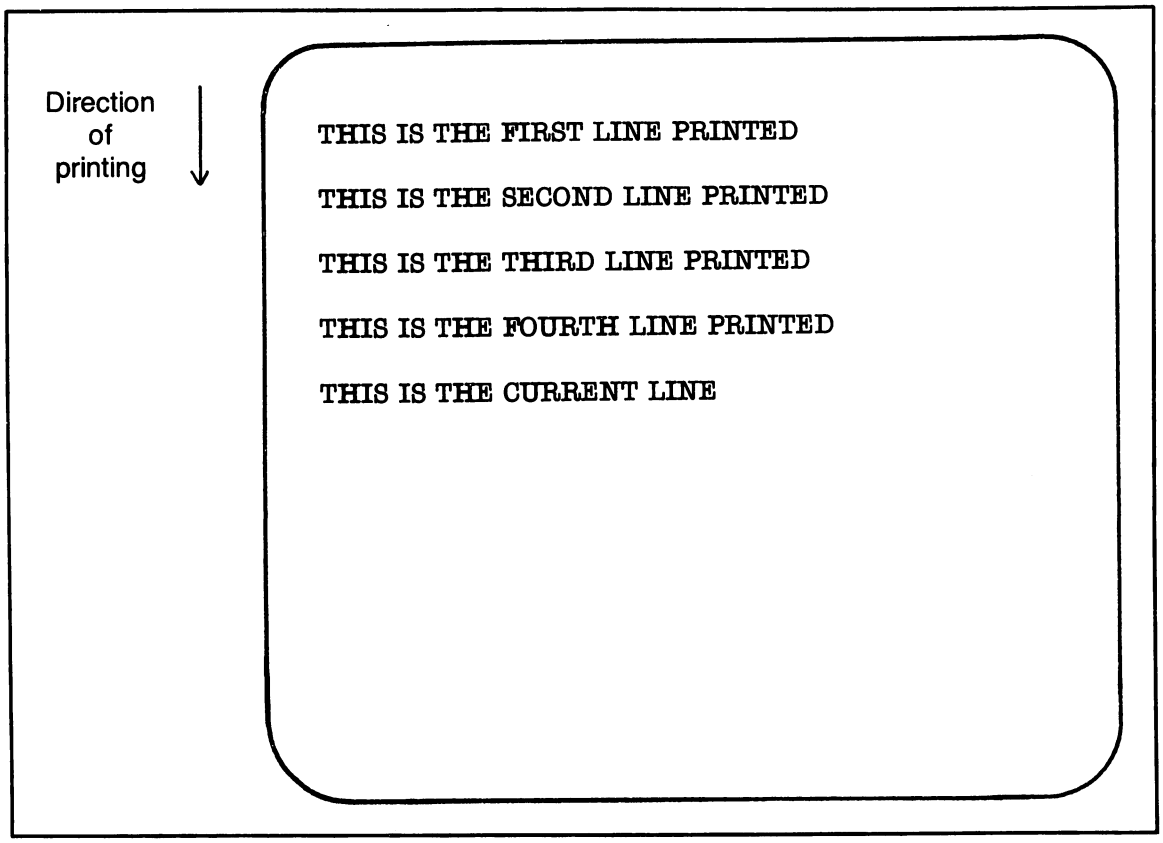


Fig. 4-3. If the screen is cleared and then a series of lines are printed, each successive line will appear one row further down the screen.

Printing CHR$(147) is equivalent to pressing the shifted CLR/HOME key).

2. Move the cursor down the required number of rows by printing CHR$(17)—equivalent to pressing CURSOR DOWN key.
3. Move the cursor across the required number of columns by printing CHR$(29)—equivalent to pressing CURSOR RIGHT key.

By combining these three statements, you can create a subroutine to move the cursor to a particular position on the screen.

Figure 4-5 contains the listing of a subroutine (lines 1260-1380) that uses these three statements to position the cursor. Subroutine arguments are:

V: Vertical position (0-24) measured in rows down from the top of the display.

H: Horizontal position (0-39) measured in columns over from the left edge of the display.

Here is how the subroutine works. Line 1270 moves the cursor to the Home position. Lines 1300-1320 contain a FOR-NEXT loop that prints CHR$(17) V times, thereby moving the cursor down V rows. Lines 1350-1370 move the cursor H spaces to the right by printing CHR$(29) H times. Line 1290 checks for V = 0 and line 1340 checks for H = 0. These lines skip over the FOR-NEXT loop if the test is satisfied to prevent the cursor movement that would occur by passing through either FOR-NEXT loop.

Lines 10-60 are not part of the subroutine,

obviously, but are included so that you can type this subroutine into your computer and try it out. When you RUN it, you will be prompted to input V and H, and after you do this and press Return, the cursor will jump to the location you indicated and print the word "HERE!", telling you where you moved the cursor.

This subroutine works reasonably well, but since it is in BASIC, it executes fairly slowly. Generating a screen character by character would be very time-consuming. In addition, since the cursor must be sent to Home each time the subroutine executes, you cannot move the cursor to a particular horizontal position without first knowing what the vertical position is. This is an inconvenience. There are various ways to get around this problem, none very convenient.

There are other ways to position the cursor using BASIC. One popular technique is to create character strings consisting of quoted CURSOR DOWN or CURSOR RIGHT keystrokes, and then to take substrings of these strings and print them. This has the effect of moving the cursor the number of spaces equal to the length of the substring. Commodore programmers have been using these and other clever tricks for many years to get the

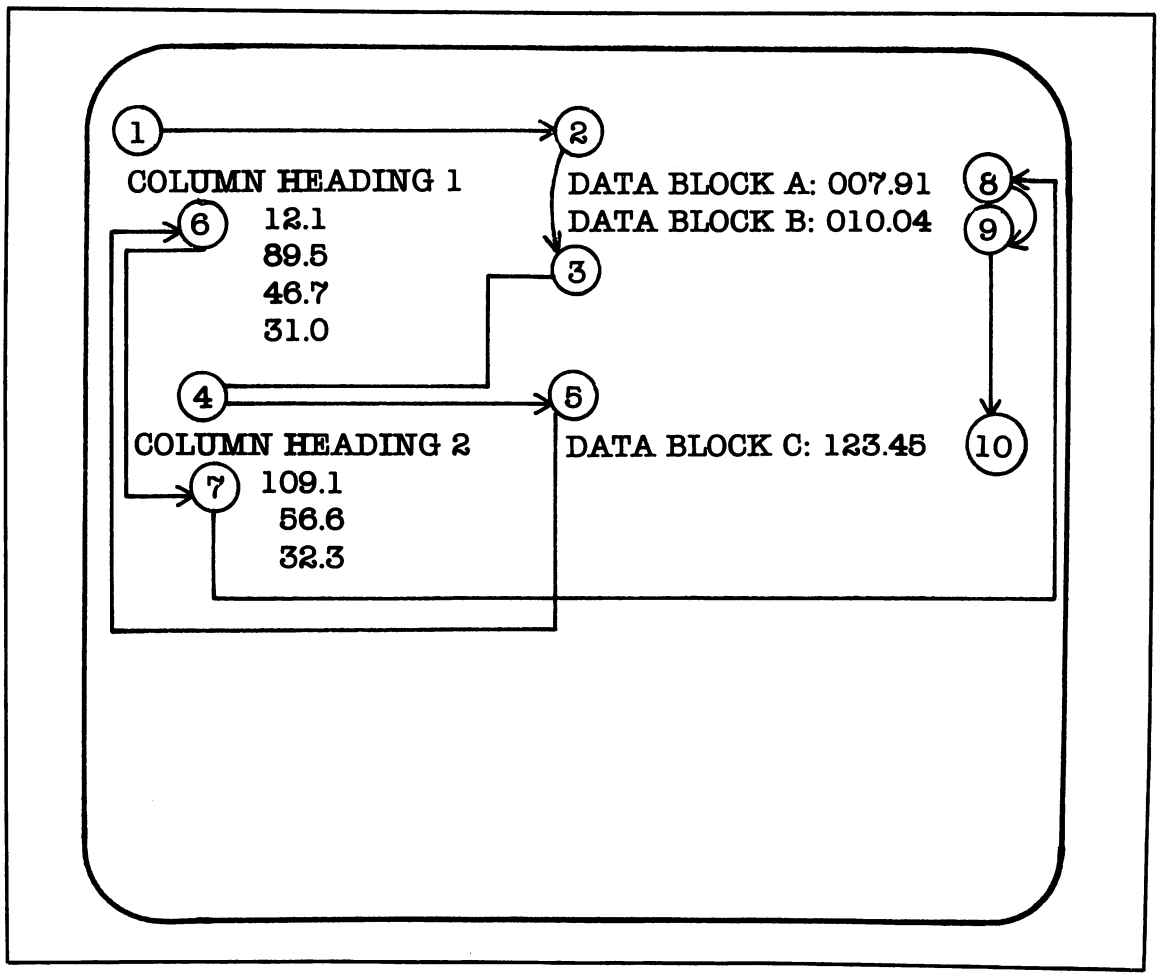


Fig. 4-4. With full cursor control, it is possible to print the different parts of the screen in whatever order is convenient, as shown here by the circled numbers, which represent the order in which the item was printed.

```
10 PRINT CHR$(147)
20 INPUT"V: ";V
30 INPUT"H: ";H
40 GOSUB 1260
50 PRINT"HERE!"
60 GOTO 20
1260 REM--CURSOR POSITIONER--
1270 PRINT CHR$(147);:REM MOVE CURSOR TO HOME
1280 REM-MOVE CURSOR DOWN V ROWS-
1290 IF V=0 GOTO 1330
1300 FOR A=1 TO V
1310 PRINT CHR$(17);
1320 NEXT
1330 REM-MOVE CURSOR RIGHT H COLUMNS-
1340 IF H=0 GOTO 1380
1350 FOR A=1 TO H
1360 PRINT CHR$(29);
1370 NEXT
1380 RETURN
```

Fig. 4-5. A BASIC subroutine (lines 1260-1380) for positioning the cursor to a particular row and column.

cursor to do what they want it to. If you want to stick with BASIC, then you can use one of these techniques.

Using Assembly Language

There is a better way, however, and it is to use assembly-language subroutines. These are faster and more powerful than what you can do with BASIC.

Before you can use these subroutines, you must first load them into memory. Figure 4-6 is the listing of the lines of code which read DATA statements and then POKE into memory the values comprising the machine-language subroutines. This listing goes from lines numbered 500-880, a range that I suggested in Chapter 3 would be good for assigning constants and reading in DATA statements. You can use a different range of lines than this—just be sure that these lines are executed at the beginning of your program.

Take time now to type in this listing and save it to disk. Type it in very carefully, and double (triple) check every value following a DATA statement. If there are errors in your version of the listing, the assembly-language subroutines will behave unpredictably. As you will see shortly, the subroutines you can access via this listing are well worth the drudgery of typing it in.

One final point. These subroutines use an area of memory from decimal 49152 up. This memory area may be used by your BASIC/DOS extension or other software and interfere with the proper use of the subroutines. Solutions are to relocate the subroutines or to deactivate the software. It is probably best to do the latter, if necessary. Later, when you have gotten the subroutines up and running, you can relocate them to another area of memory.

Relocation is very simple. Line 510 contains the statement C0 = 49152. This is the starting point in memory of the subroutines. Various values are POKEd into this decimal location and into higher locations in memory until line 540 READs a value of D = 0, which signals the end of the DATA statements (see line 870).

Now, let us try out an assembly-language subroutine. You call one of these subroutines with the SYS statement, followed by the subroutine's parameters. The syntax is as follows:

SYS C0,OPCODE,parameter 1, parameter 2.

C0 is the memory location (decimal) at which the machine language subroutine is accessed, that

is, decimal location 49152.

OPCODE is a value between 0 and 4 that tells the computer which type of subroutine you are calling. The OPCODE for the cursor-positioning subroutine is 0.

With an OPCODE of 0, parameter 1 is the cursor's vertical position (1-24), measured down from the top of the display (Fig. 4-7). Parameter 2 is the horizontal position (1-40), measured over from the left edge of the display (Fig. 4-7).

Putting together these pieces, a call for the cursor-positioning subroutine looks like this:

SYS C0,0,V,H

V is the cursor's vertical position and H is its horizontal position. For example, to move the cursor to row 15 and column 23, use the statement SYS C0,0,15,23 in your program.

Do not take my word for it. Add the lines

```
500 REM--LOAD ASSEMBLY LANGUAGE SUBROUTINES--
510 C0=49152
520 READ B
530 POKE C0,D:C0=C0+1
540 IF D=0 THEN READ D:POKE C0,D:C0=C0+1:IF D=0 GOTO 880
550 GOTO 520
560 DATA 173,72,160,133,20,173,73,160
570 DATA 133,21,160,6,169,96,153,232
580 DATA 3,177,20,153,231,3,136,208
590 DATA 248,160,3,169,0,153,252,3
600 DATA 136,16,250,32,121,0,201,0
610 DATA 240,26,201,58,240,22,230,122
620 DATA 208,2,230,123,200,132,251,32
630 DATA 232,3,164,251,165,20,153,252
640 DATA 3,184,80,223,24,173,252,3
650 DATA 42,42,42,42,42,42,144,1
660 DATA 42,141,252,3,56,32,240,255
670 DATA 173,253,3,240,2,170,202,142
680 DATA 253,3,173,254,3,240,2,168
690 DATA 136,140,254,3,192,40,176,30
700 DATA 173,253,3,201,26,176,23,169
710 DATA 255,44,252,3,112,23,8,174
720 DATA 253,3,172,254,3,24,32,240
730 DATA 255,40,240,110,80,7,169,161
740 DATA 141,204,5,208,101,16,5,169
750 DATA 25,141,255,3,173,255,3,208
760 DATA 6,173,253,3,141,255,3,205
770 DATA 253,3,144,226,201,26,176,222
780 DATA 173,254,3,201,40,176,215,206
790 DATA 255,3,169,0,133,251,169,4
800 DATA 133,252,174,253,3,240,16,24
810 DATA 165,251,105,40,133,251,165,252
820 DATA 105,0,133,252,202,208,240,172
830 DATA 254,3,169,32,145,251,200,192
840 DATA 40,144,247,152,160,0,24,101
850 DATA 251,133,251,144,2,230,252,238
860 DATA 253,3,173,255,3,205,253,3
870 DATA 176,224,96,0,0
880 C0=49152
```

Fig. 4-6. Code for loading several assembly-language subroutines important for screen control.

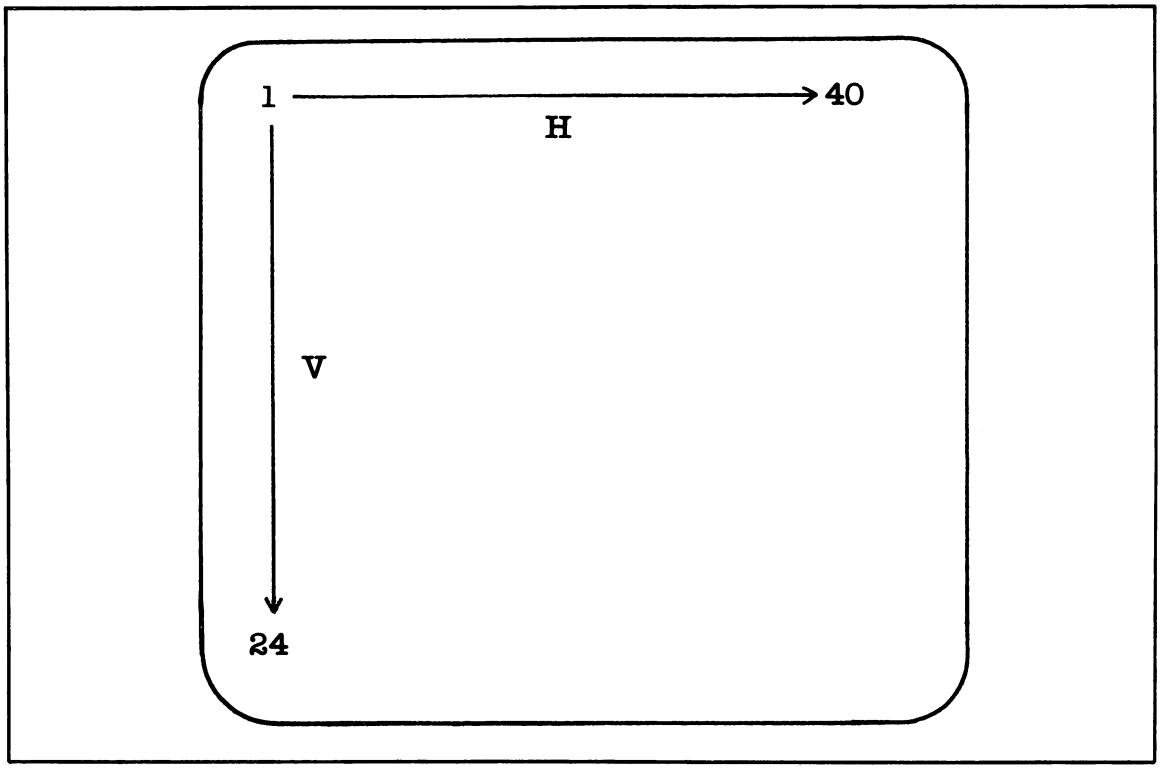


Fig. 4-7. SYS C0,0,V,H can be used to position the cursor to row V and column H of the display.

shown in Fig. 4-8 to your loading program (Fig. 4-6). The added lines will allow you to set the OPCODE, and then enter parameters 1 and 2 to position the cursor at various locations on the display.

Incidentally, the input routines in Fig. 4-8 are of the "quick and dirty" variety that programmers write for themselves to try things out. There is no error-testing, and you can make them crash. They are all right for the sort of informal use recom-

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
10000 REM--MACHINE LANGUAGE SUBROUTINE DEMONSTRATION PROGRAM--
10010 PRINT CHR$(147)
10020 INPUT"OPCODE: ";OP
10030 INPUT"PARAMETER 1: ";P1
10040 INPUT"PARAMETER 2: ";P2
10050 SYS C0,OP,P1,P2
10060 PRINT"*"
10070 GOTO 10030
```

Fig. 4-8. This program (with the subroutine loader) can be used to try various OPCODEs and SYScalls.

```
10 REM ************************************
20 REM * NOTE: THIS SUBROUTINE REQUIRES   *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
1260 REM--CURSOR POSITIONER--
1270 SYS C0,0,V,H
1280 RETURN
```

Fig. 4-9. Subroutine (lines 1260-1280) that calls assembly-language subroutine to position cursor.

mended, but no model for how to write code for use in an actual program. In that case, you must take much more care to prevent incorrect or accidental inputs from killing your program. More on that in Chapter 5.

After you have typed in the lines in Fig. 4-8, SAVE the program to disk with a different name than you used earlier. Then RUN it. If your computer hangs up, make sure that none of your software is using the required memory area. If it is, do a cold start, reload the program, and start over.

You are ready to try the routines when the program prompts you to type in the OPCODE. Type in the number 0 and press RETURN.

You will then be prompted to enter PARAMETER 1 and then PARAMETER 2. Type in values in the legal range and press the Return key. If the routine is working properly, a star (*) will print at the location you specify. If the routine does not work properly, the most likely cause is an incorrect value following one of the DATA statements in the listing shown in Fig. 4-6. If you have a problem, check your listing against the original again. If all works well, then play around with the routine until you feel comfortable with it. Try various values of V and H to see what happens.

Since this is an assembly-language subroutine, it is much faster than BASIC. It also gives you more flexibility. You can use a SYS statement directly in code, or put it into a subroutine. Since a SYScall is so short, usually it is simple enough to use directly in code.

Figure 4-9 is the listing of a subroutine based on the SYScall just described. Note that this subroutine starts at line 1260, the same line as the BASIC subroutine described earlier and shown in Fig. 4-5. The assembly-language version is considerably shorter—and much faster.

To illustrate how the subroutine (or direct SYScalls) may be used to generate a screen, suppose that you have laid out the screen shown in Fig. 4-10, and want to write a little program that will generate the screen by first printing the headings, and then going back and printing the data.

Figure 4-11 shows code that will generate the screen in these two stages. The headings are printed starting at line 11000, and the data are printed starting at line 12000.

Consider what would be involved in generating a screen like this on a line-by-line basis, starting at the top of the display—it is much easier to print the headings first and then go back and print the data.

One nice thing about the cursor-positioning subroutine is that it gives you greater control of the cursor for horizontal tabbing. The C-64 TAB statement allows you to print information at a particular horizontal tap position, but it does not always follow orders. To illustrate, type in and RUN the following program:

```
10  PRINT TAB (10) "AAAA";
20  PRINT TAB (12) "BBBB"
```

The output is two strings printed end to end: AAAABBBB. Your program told the computer to print AAAA at the tab position 10 and BBBB at tab position 12. The second string should have overwritten the first.

What happened? C-64 BASIC retained the integrity of the two strings by printing them end to

end instead of overprinting them. It helped you out, even though you did not ask it to. I do not want to anthropomorphize this too much, but I can imagine BASIC saying something to itself such as, "That poor ninny couldn't possibly mean for those two strings to print on top of each other! I'll fix things by moving the second one over a few spaces." Assume however, that you knew what you were doing (a reasonable assumption) and wanted to overprint them? Too bad. You cannot do it with the TAB statement. The C-64's TAB is not unique in this respect. Most TABs work this way.

Another thing that TAB does not permit is to print the columns in a row from right to left instead of from left to right. For example, modify your program so that it looks like this:

```
10  PRINT TAB (30) "AAAA";
20  PRINT TAB (10) "BBBB"
```

Now RUN it. The output is still AAAABBBB, but starting at column 30. The TAB (10) statement in line 20 was ignored. BASIC would not permit you to move the cursor backward with TAB.

You do not have these problems with SYScalls. To illustrate, add these lines to your version of the

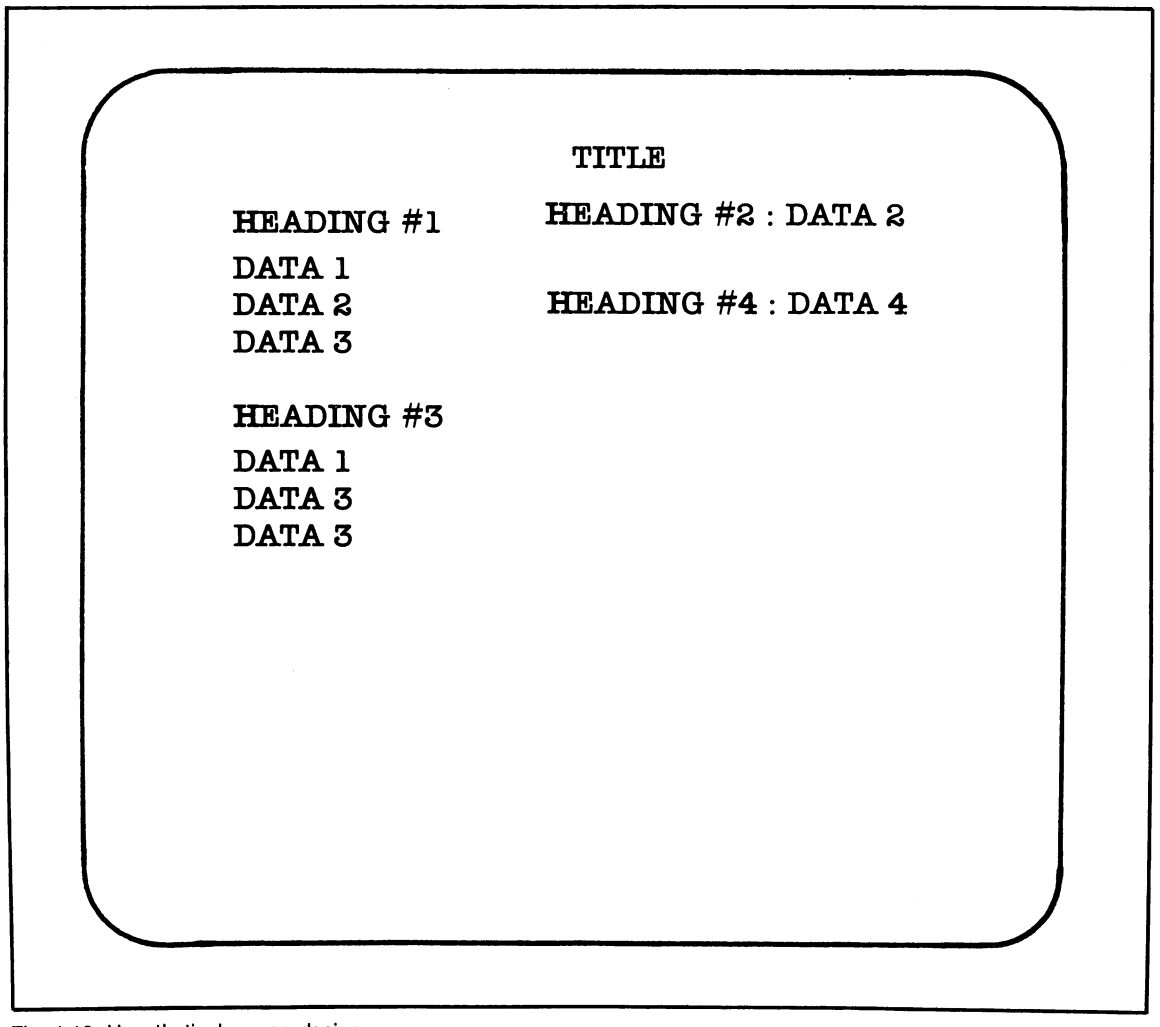


Fig. 4-10. Hypothetical screen design.

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
10000 REM---GENERATE DISPLAY---
10010 REM--PRINT TITLE--
10020 PRINT CHR$(147):REM CLEAR SCREEN
10030 SYS C0,0,1,18
10040 PRINT"TITLE"
11000 REM--PRINT HEADERS--
11010 SYS C0,0,3,3
11020 PRINT"HEADING #1"
11030 SYS C0,0,3,16
11040 PRINT"HEADING #2"
11050 SYS C0,0,12,3
11060 PRINT"HEADING #3"
11070 SYS C0,0,6,16
11080 PRINT"HEADING #4"
12000 REM--PRINT DATA--
12010 REM-DATA 1-
12020 FOR V=5 TO 7
12030 SYS C0,0,V,3
12040 PRINT"DATA 1"
12050 NEXT
12060 REM-DATA 2-
12070 SYS C0,0,3,28
12080 PRINT"DATA 2"
12090 REM-DATA 3-
12100 FOR V=14 TO 16
12110 SYS C0,0,V,3
12120 PRINT"DATA 3"
12130 NEXT
12140 REM-DATA 4-
12150 SYS C0,0,6,28
12160 PRINT"DATA 4"
12170 END
```

Fig. 4-11. Code for generating screen shown in Fig. 4-10 by first printing headings and later printing the data.

routine shown in Fig. 4-6 or 4-8:

```
10010  SYS C0,0,10,10
10020  PRINT"AAAA";
10030  SYS C0,0,10,12
10040  PRINT"BBBB".
```

RUN the program and watch what happens. The output is AABBBB, starting at column 10. This routine does not second-guess you, and permits you to overprint if you have a mind to.

You can also print backwards. To illustrate, change lines 10010 and 10030 to read as follows:

```
10010  SYS C0,0,10,30
10030  SYS C0,0,10,10
```

RUN the program. AAAA appears starting at column 30 and BBBB starting at column 10—you were able to print backwards!

CLEARING A LINE OR RANGE OF LINES

In generating screens, it is often necessary to clear part of the display. You can clear the C-64's entire display by printing a CHR$(147), but there is no simple way to clear just part of the display. It is no fun to erase and rewrite an entire screen when you only need to change one or two parts of it. Not only is this extra work, but it is irritating to the user, whose displays periodically disappear and then return.

One method to clear part of the display is to print a long string of blanks. For example, you can define this string early in your program:

BLANK$="   "

Once BLANK$ has been defined, you can use it to print over unwanted information. This looks like a solution, but unfortunately, it is not. C-64 BASIC gets confused when you print several rows of blanks, and doublespaces between subsequent lines that it prints.

Fortunately, the assembly-language routine described in the previous section (see Fig. 4-6) can be used to clear part or all of your display. The various clearing options follow.

Clearing a Single Line

You can clear part or all of a single line by using an OPCODE of 1. The syntax of the line-clearing subroutine is as follows:

SYS C0,1,V,H

This SYScall clears to the end of a line from cursor location V (vertical position) and H (horizontal position).

Examples:

SYS C0,1,22,20 clears columns 20-40 of row 22
SYS C0,1,4 clears all of row 4

Use this call directly in code or in a subroutine. Lines 1300-1320 of Fig. 4-12 are the subroutine form of this SYScall.

Clearing to End of Screen

You can clear from the cursor position to the end of the screen by using an OPCODE of 3. The syntax of the to-end-of-screen clearing subroutine is as follows:

SYS C0,3,V,H

This SYScall clears to the end of the screen from cursor location V (vertical position) and H (horizontal position).

```
10 REM ************************************
20 REM * NOTE: THESE SUBROUTINES REQUIRE *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE    *
40 REM * LOADER (FIGURE 4-6)             *
50 REM ************************************
1300 REM--CLEAR ONE LINE--
1310 SYS C0,1,V,H
1320 RETURN
1350 REM--CLEAR TO END OF SCREEN--
1360 SYS C0,3,V,H
1370 RETURN
1400 REM--CLEAR RANGE OF LINES--
1410 SYS C0,4,V1,0,V2
1420 RETURN
```

Fig. 4-12. Subroutines that call assembly-language routines to (a) clear one line, (b) clear to the end of screen, and (c) clear a range of lines.

Examples:

SYS C0,3,1,1 clears the entire screen.
SYS C0,3,15 clears everything from row 15 down.
SYS C0,3,15,20 clears everything to the right of column 20 in row 15, and all of rows 16-25.

Use this call directly in code, or in subroutines. Lines 1350-1370 of Fig. 4-12 are the subroutine form of this SYScall.

Clearing a Range of Lines

You can clear a range of lines by using an OPCODE of 4. The syntax of the range-of-lines clearing subroutine is as follows:

SYS C0,4,V1,0,V2

This SYScall clears all lines between vertical locations V1 (top vertical position) and V2 (bottom vertical position).

Examples:

SYS C0,4,1,0,10 clears rows 1 through 10.
SYS C0,4,15,0,17 clears rows 15 through 17.

Trying the SYSCalls Out

The SYScalls illustrated for positioning the cursor and for clearing parts of the screen are used frequently in the rest of this book. They are useful tools with which you should become familiar. The best way to do so is to try them out in programs. If you do not do this, you may eventually begin to feel like the college freshman who cut the first four weeks of an organic chemistry class, and later tried to cram everything in the night before the first midterm. Far better to get the hands dirty early with the necessary chemicals.

Figure 4-8 contains everything you need to demonstrate these subroutines yourself. If you did not type that listing in earlier and try it, do so now.

RUN the program and try various OPCODES and arguments. Note that, in order to use an OPCODE of 4 (for clearing a range of lines), you must modify line 10050 and include a 0 between parameters P1 and P2. That is, line 10050 should read as follows:

10050 SYS C0,OP,P1,0,P2

The next step beyond using the demonstration program is to build your own program by combining the routine for loading the assembly-language subroutine (Fig. 4-6) with your own SYScalls or perhaps with the subroutine versions of these calls shown in Fig. 4-11 and 4-12. Go for it!

After you have played with these routines for a while, devise your own variations and see what happens. To make this interesting, these routines can do some things that I have not told you about. Hint: all OPCODES between 0 and 4 are legal. If you find anything really interesting, write me a letter.

SCREEN ACCESS

Have you ever worked on a time-sharing computer system? If you have, then most of what the computer displayed to you came scrolling off the bottom of the display. Walk into a computer science laboratory sometime and watch the displays as the students work with their computers. The screens are generated line by line, and go scrolling up. If you watch long enough, it can make you dizzy. If you have vertigo, you may be in trouble.

People who started their computer experience with time-sharing systems, large mainframe computers, or the venerable Teletype are used to scrolling screens, and often take them for granted. If you started your computer experience with a microcomputer, however you may have an altogether different perspective. Most well-written programs avoid scrolling screens and use the paging technique. By "paging" I mean (1) clearing the screen, and (2) generating the entire screen at once, from top to bottom. Paging makes a clear break between the old display and the new. It also causes less eyestrain.

Scrolling causes eyestrain, shows a fixation

```
10 REM ************************************
20 REM * NOTE: THIS SUBROUTINE REQUIRES  *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE    *
40 REM * LOADER (FIGURE 4-6)             *
50 REM ************************************
4010 REM--TEMPORARY PAUSE--
4020 SYS C0,0,24,6
4030 PRINT CHR$(18);:REM REVERSE VIDEO
4040 PRINT "[PRESS SPACE BAR TO CONTINUE]";
4050 GET A$
4060 IF A$<>" " GOTO 4050
4070 PRINT
4080 RETURN

READY.
```

Fig. 4-13. Subroutine (lines 4010-4080) to print the prompt "[PRESS SPACE BAR TO CONTINUE]" and halt the program until the space bar is pressed.

on the single line (rather than a more healthy orientation toward the entire screen), and, for all we know may be a sign of poor character (add anything else that fits). Well, perhaps it is not that bad, but it is not a good thing.

It is easy to page. Just clear the display before you create the screen. Clear the display by printing CHR$(147) at the beginning of your screen-generating code. This is equivalent to pressing the shifted CLR/HOME key for the keyboard. It erases the display and moves the cursor to the upper left corner of the display. Put a semicolon at the end of the Print CHR$(147) statement or the cursor will move down to row 2 of the display. Once you have cleared the display, create the screen by PRINTing its content.

Most display screens require some overt action from the operator before they are replaced. Some programs—of the demonstration variety—display a screen for fixed amount of time, such as 30 seconds, before erasing it and displaying the next screen. It is usually necessary to introduce a pause into the program after displaying the screen. This may be done in several ways. One common way is to require user input. Inputs are discussed in Chapter 5.

Another common way is to introduce a temporary pause that requires the user to press the space bar or a function or other key. Figure 4-13 is the listing of a simple subroutine that prints the following statement in reverse video, centered and at row 24 of the display, awaiting a space bar press:

[PRESS SPACE BAR TO CONTINUE]

This subroutine uses the assembly-language cursor-positioning subroutine presented in Fig. 4-9 of this chapter. You can modify the subroutine to look for some other key press by changing what you put between the quotation marks in line 4060.

Using this subroutine is simple. It has no arguments. Just insert a GOSUB 4010 whenever you want a temporary pause in your program. When the subroutine executes, it displays the message and awaits the appropriate key press before it continues.

Another way to pause is to insert a time delay of a particular number of seconds. Figure 4-14 is the listing of a subroutine that introduces a delay of TX seconds (more or less). You might use a delay such as this in a series of screens that are presenting instructions or a demonstration to the user. Adjust

```
4100 REM--TX SECOND DELAY--
4110 FOR A=1 TO TX*870
4120 NEXT
4130 RETURN
```

Fig. 4-14. Subroutine to introduce a delay of TX seconds.

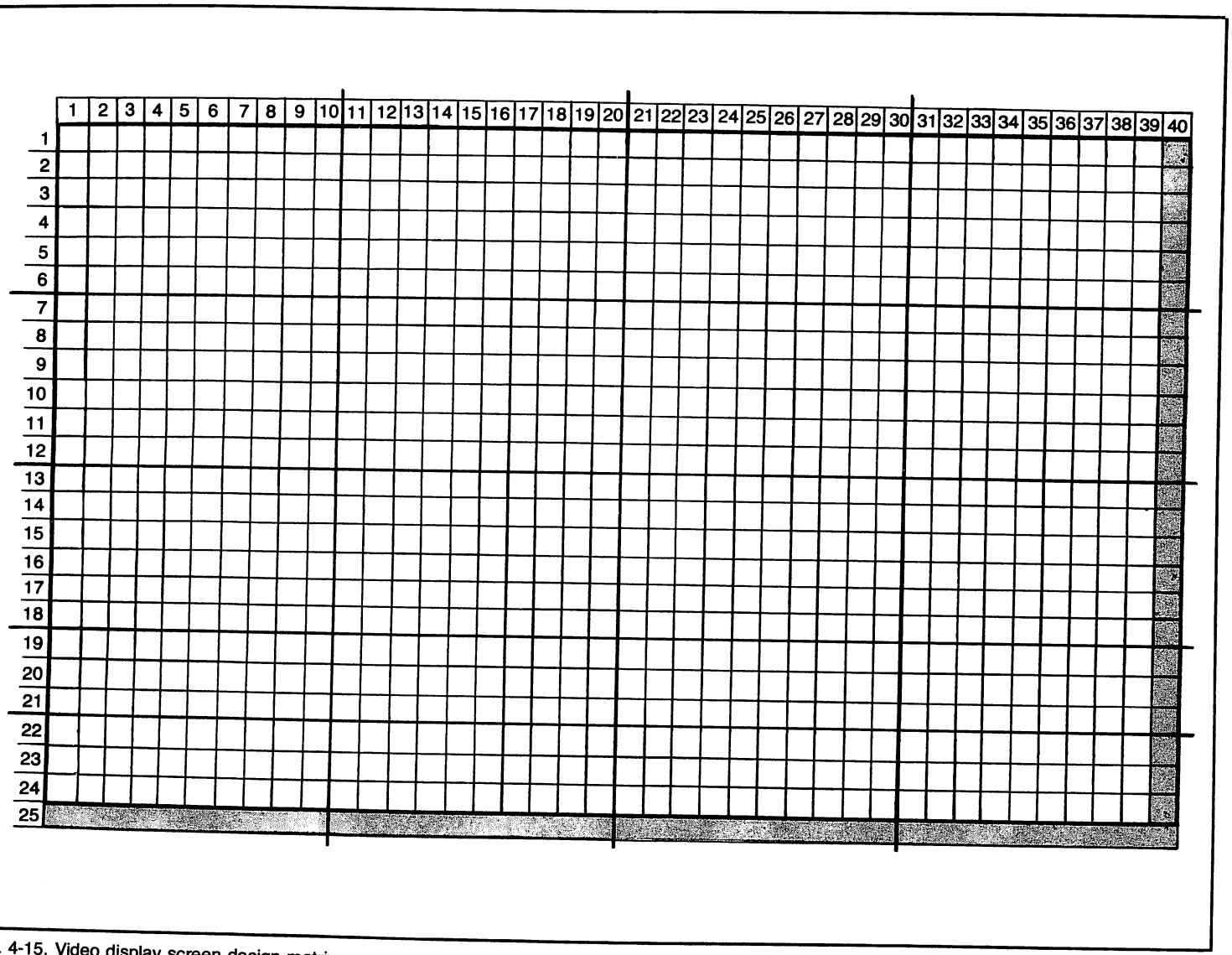


Fig. 4-15. Video display screen design matrix.

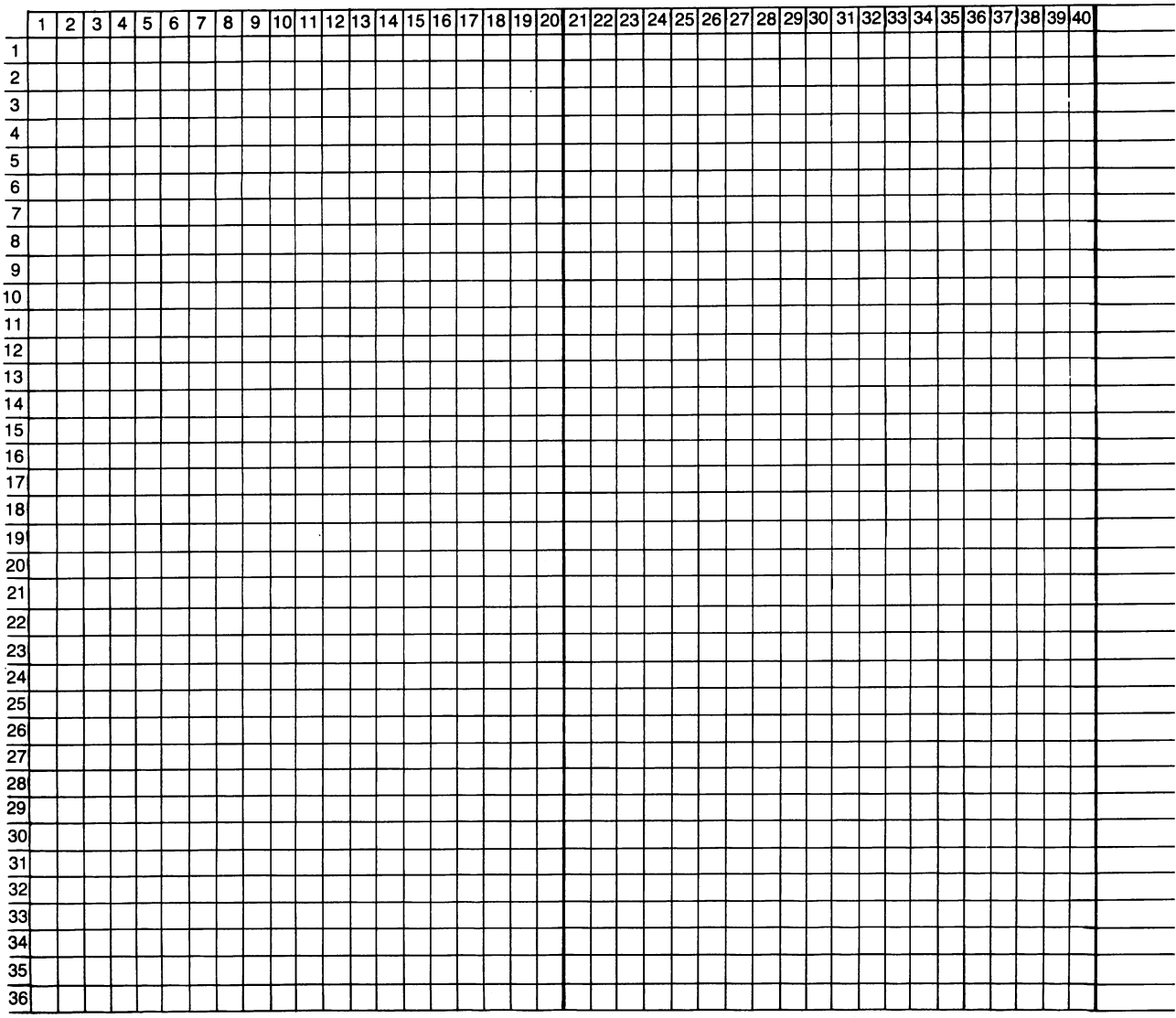
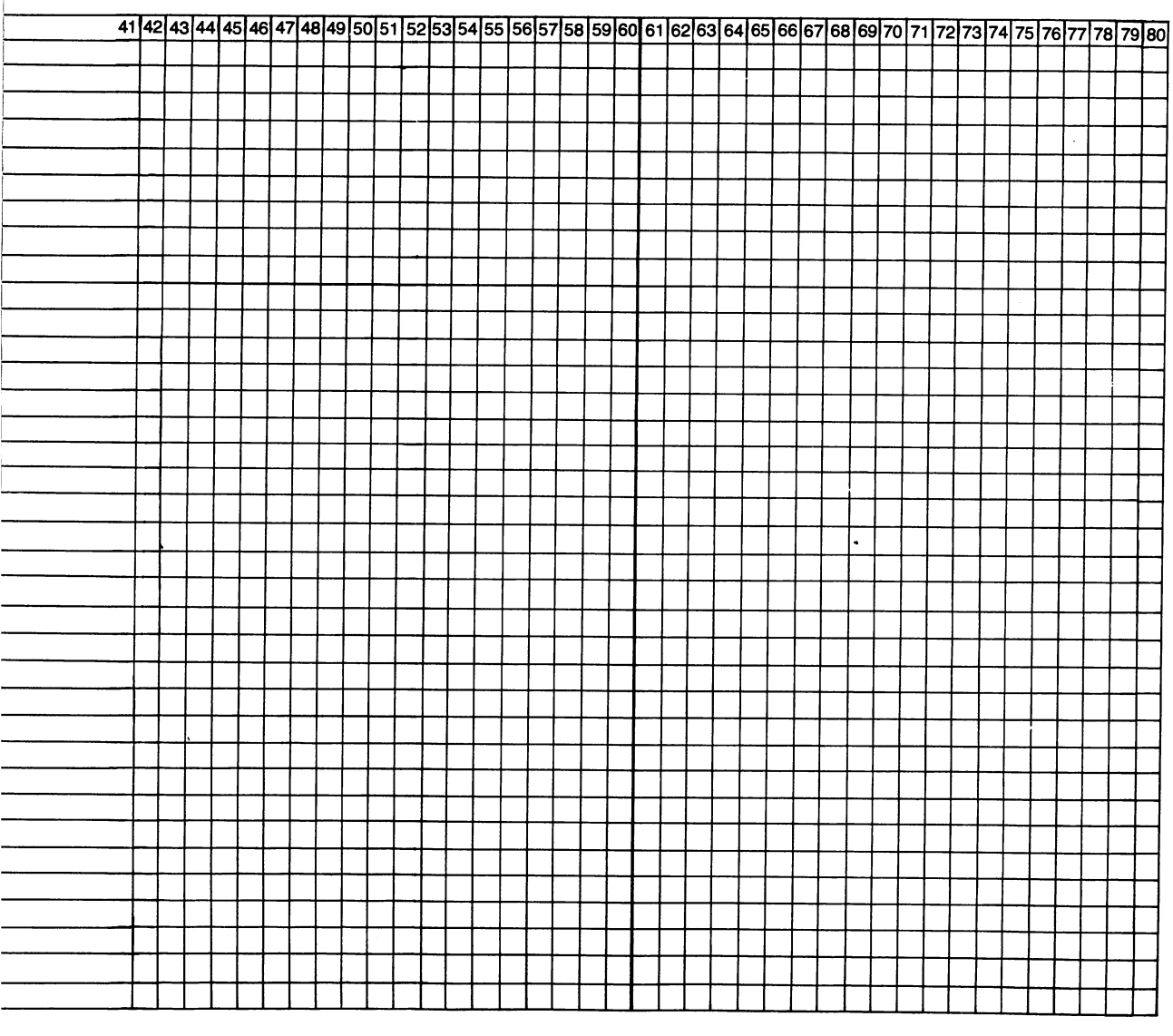


Fig. 4-16. Printed report design matrix.

the display time to suit the subject. (If the information is difficult, let the user control how long it is displayed.) Usually 30 seconds is long enough (or too long!) to digest a screen. To introduce a delay this long, insert in your program the statements TX=30 and GOSUB 4100. When the subroutine executes, the FOR-NEXT loop in lines 4110-4120 will start counting and distract your computer's attention for about 30 seconds before RETURNing.

HOW TO LAY OUT A SCREEN

Display screens can be well or poorly designed. Knowing a few simple rules can improve your design a great deal. This section offers some of these rules, along with several examples. Rules are relative, of course. You are an intelligent, thinking being, or you would not be interested in computers or in reading this book. Breaking these rules will not land you in jail, but it will make your screens



less understandable and attractive.

Still, it sometimes makes sense to break the rules. As they always say, however, before you break a rule, know what it is.

Use a Screen Design Matrix

Figure 4-15 is a screen design matrix. They are wonderful things. You can use them to design a screen with paper and pencil before you sit down to your computer and start coding.

Columns (1-40) are listed across the top and bottom, and rows (1-25) down the left and right sides. Note that the right column (40) and the bottom row (25) are solid black to remind you not to print anything in them. You can, of course, but doing so complicates screen generation. Whenever you print something in column 40 or row 25, BASIC

volunteers an extra line feed, which either causes an extra space or causes the screen to scroll.

Creating a screen is an act of planning and design which should be done with paper and pencil, not sitting at your computer attempting to get the display to come out right. Coding should be a mechanical act to translating the paper plan into cursor-positioning and PRINT statements.

Take Fig. 4-15 to your copy shop and make several copies of it for use later. Do the same with Fig. 4-16, the 80-column version, which is used for laying out hard-copy report formats.

Title Your Display

How would you feel if you walked into a bookstore whose books had blank front covers? How about going to a convention where all the folks had little paper signs on their pockets that said:

HI!

MY NAME IS _____

The problem is simple. People want to know with what or whom they are dealing. Tell them. Even if it is obvious to you, there are many to whom it is not obvious. Even experienced user sometimes becomes disoriented—you know, they get up from the computer, walk into the other room, look at a football game or talk to someone—and when they return, they have lost track of what display they were looking at earlier.

Solution: Label each display by printing a title at the top of it. I suggest that you put that title in all capital letters and reverse video.

Figure 4-17 is the listing of a subroutine that centers and prints any title in reverse video. The argument of the subroutine is T$ (title). Set T$ equal to the title and then call this subroutine. For example, you can center and print the word "TITLE," by including these lines in your program:

```
100 T$="TITLE"
110 GOSUB 1500
120 END
```

```
1500 REM--CENTER & PRINT T$--
1510 T=(39-LEN(T$))/2
1520 PRINT CHR$(18);
1530 PRINT TAB(T) T$
1540 RETURN
```

Fig. 4-17. Subroutine to center and print title on display screen.

When these lines are executed, they center and print the title in reverse video.

The subroutine does not clear the screen first; so you can use it to center and print a title anywhere on the screen by first locating the cursor to the correct row. If you do want to print the title at the top, clear the screen by printing a CHR$(147) before calling the subroutine.

Divide the Screen into Logical Areas

The display screens that you use in programs will probably contain certain classes of information. For example:

- Screen title—Centered at the top of the display.
- Data—Text, numbers graphics, or some combination of all three. Data are usually printed on the center of the display.
- Data input area—Input prompts may appear at the top, center, or bottom of the display.
- Control area—A menu or control prompt may be printed anywhere on the display.
- Message area—Warnings, status information, or other messages may be printed anywhere on the display.

These are the most common categories of information that appear on display screens. The only thing fixed about where information should be displayed is that the title should appear at the top and data should appear below it. Other than this, you can design a screen display any way that makes sense to program users.

Most programs use several different display screens. These screens contain some or all of the

classes of information just mentioned, as well as others not noted. The rule of consistency argues for presenting each class of information on a particular part of the display. To ensure that your displays are consistent, analyze what types of screens you need in your program. Determine what classes of information you must present. Then lay out one or more general screen plans for use as models in building your screens. Find the smallest number of screen plans—ideally one—that you can use throughout your program.

To illustrate, Figs. 4-18 through 4-20 show screen plans for a hypothetical program. Screen 1 (Fig. 4-18) is the main control menu screen. This program uses several control menus. All are formatted according to this model. The title appears at the top, menu options at the center, and the prompt line at the bottom.

Screen 2 (Fig. 4-19) is the data entry screen. The title appears at the top. Numbered data entry fields are at the center. The bottom contains a menu for adding, modifying, or deleting entries, or for exiting the program. The message line also appears at the bottom.

Screen 3 (Fig. 4-20) is the display screen for presenting numeric results. The title appears at the top. Subtitle, column headings, and numeric information fill the center. The bottom contains a control menu for selecting a new screen or exiting the current screen.

The models you develop depend upon your program. Follow the models as you code the program to ensure consistency among your screens.

Of course, once you sit down and start coding your program, there may be exceptions to each model. Do not force things to fit. Consistency is a good thing to keep in mind, but the primary goal of any display is to do a job—convey information, permit control of the program, take data entries, or whatever.

When a screen contains more than one type of information, label the different types and separate

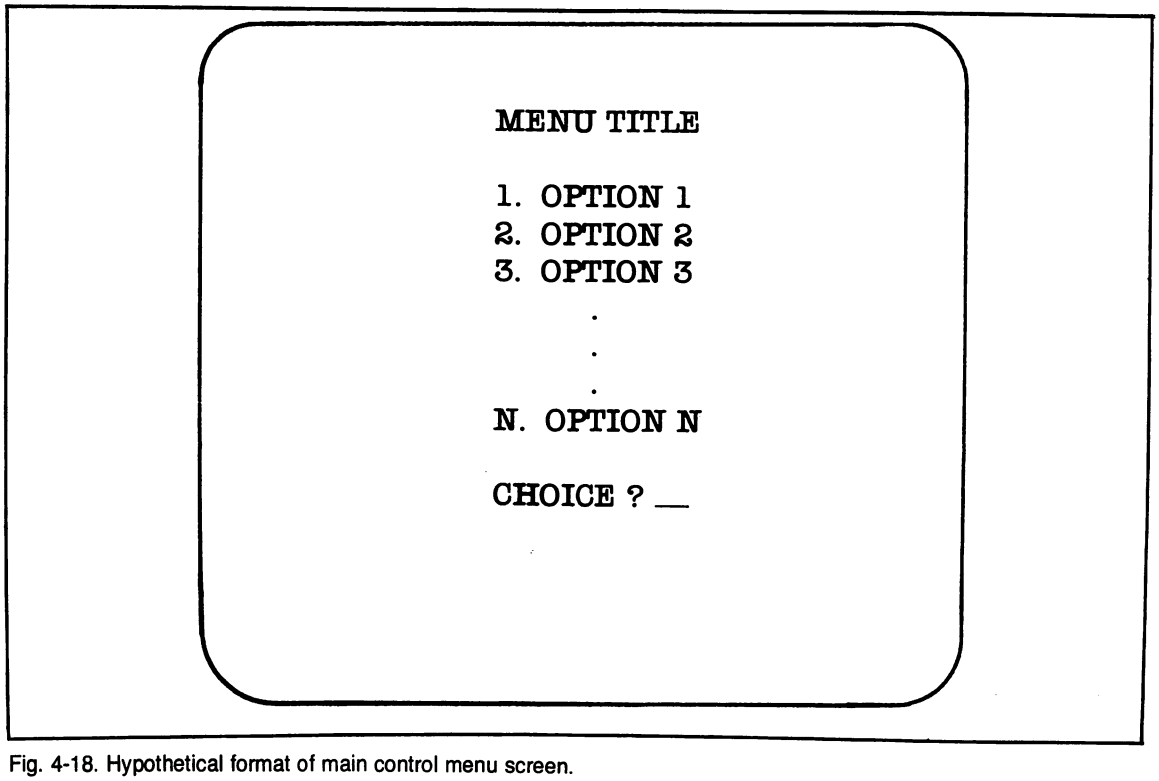


Fig. 4-18. Hypothetical format of main control menu screen.

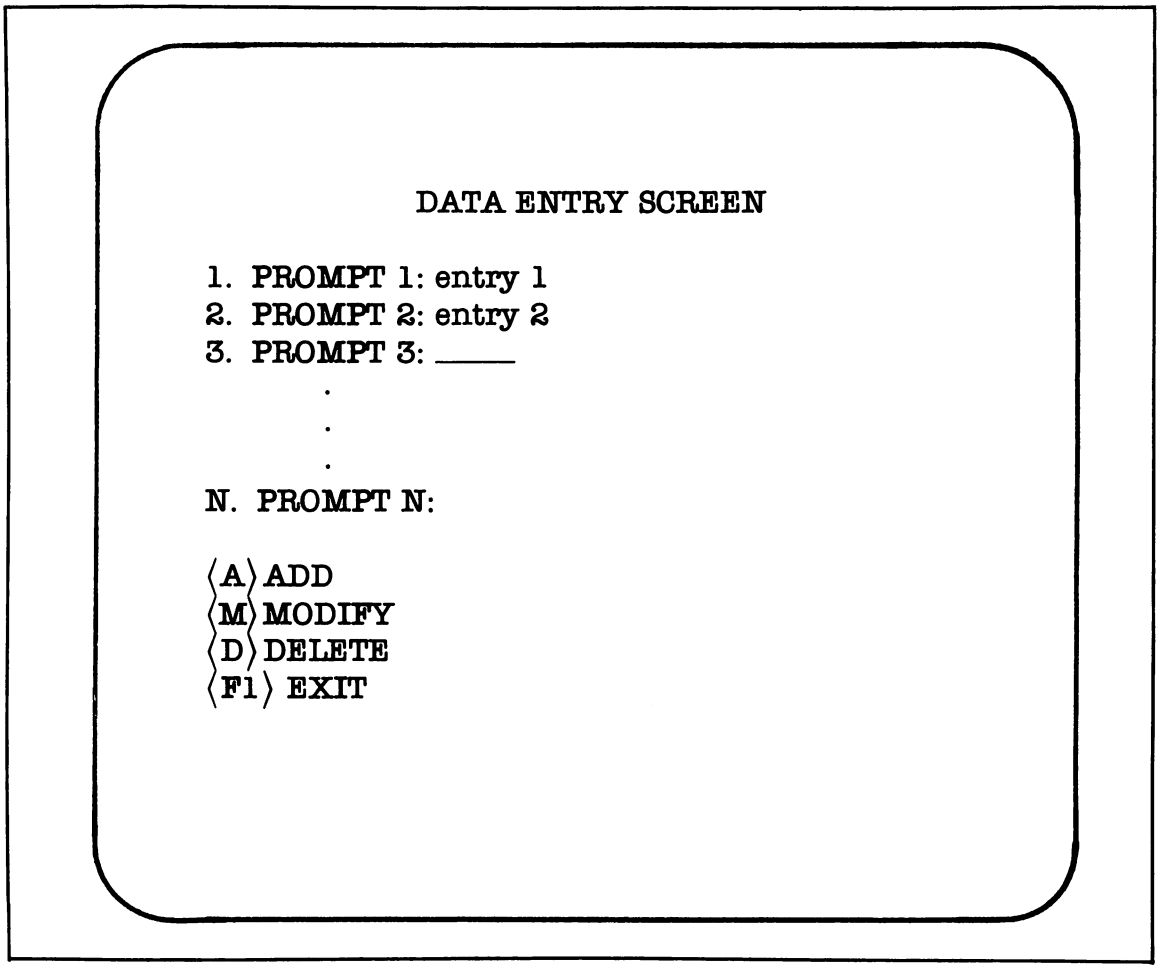


Fig. 4-19. Hypothetical format of data-entry screen.

them from one another. If you do not do this properly, the user may have difficulty making sense of the screen. The best way to illustrate this point is by example. Figure 4-21 is an example of a poorly designed display screen. Here is what is wrong with it:

- The display has no title.
- There are three different data areas, but they overlap.
- The column headings are meaningless abbreviations.
- The prompt at the bottom of the screen is ambiguous.
- Headings and prompts are difficult to separate from data.

Figure 4-22 shows a better way to design this screen. Here are the differences:

- The display is titled.
- The center of the screen is divided into three distinct data areas, separated by dashed lines. Each area is separately titled.
- Column headings are complete words, not abbreviations.
- The prompt gives explicit directions.

- Headings and prompts are in reverse video to make them stand out.

A few changes improve the screen a great deal.

Using reverse video for titles, headings, and prompt lines is optional, but these three items must stand out. Instead of using reverse video, the items could be underlined or enclosed in boxes.

Avoid abbreviations if you can. Many programmers have the mistaken idea that abbreviations are expected on computer displays, and they are easy to understand. On the first point, they may be correct, but on the second wrong. Abbreviations are harder to understand. For this reason, avoid them.

The data area may be divided in a number of different ways. In Fig. 4-22, lines were printed between areas to separate them. The areas could also be separated by leaving at least two extra spaces.

A third way is to use color—to divide the screen up into blocks and print each block with a different background and character color. While this is more difficult to program, you do not need as much space between screen areas because they are clearly separated by their colors. When you do use color this way, be sure that the different color blocks have approximately the same brightness levels so that the eye does not have to adapt to different light levels within a single display. The

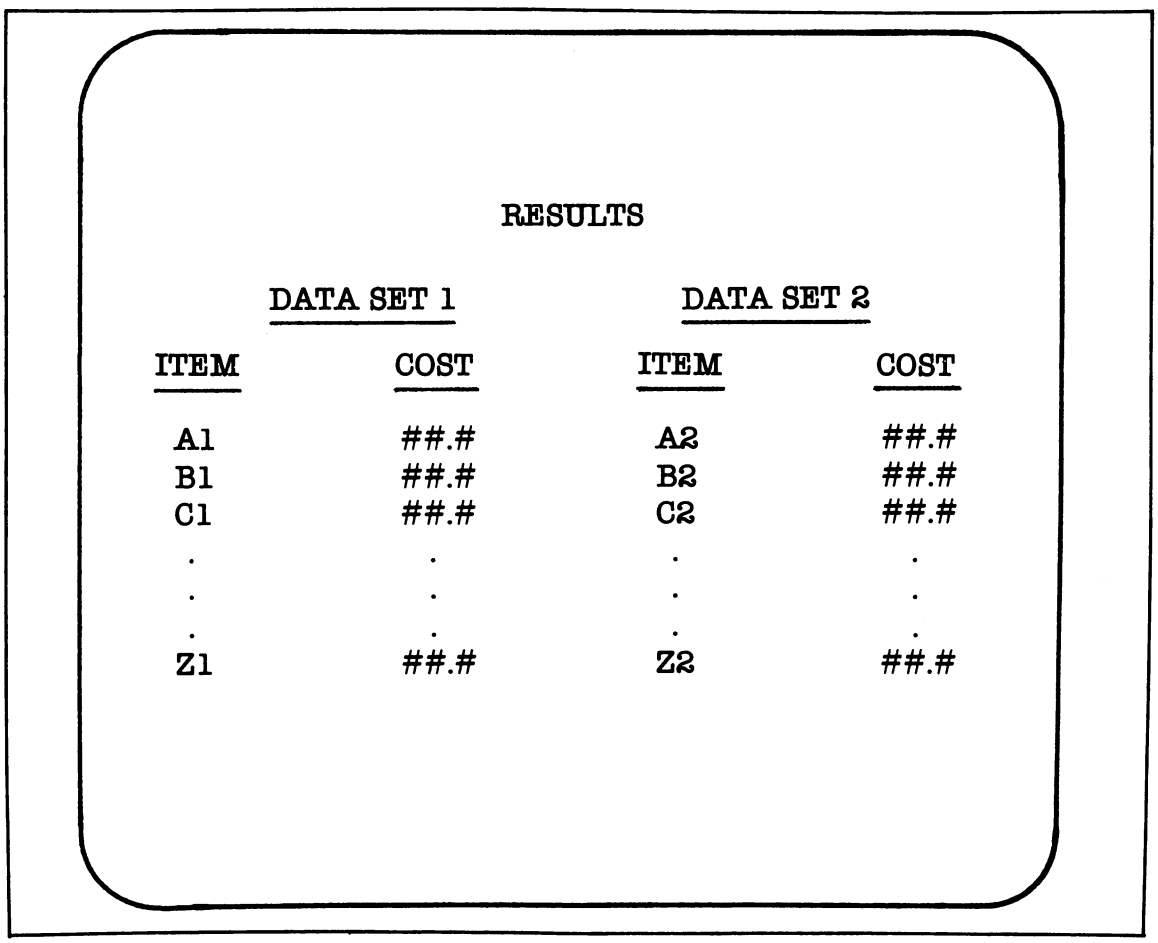


Fig. 4-20. Hypothetical format of display screen for presenting numerical results.

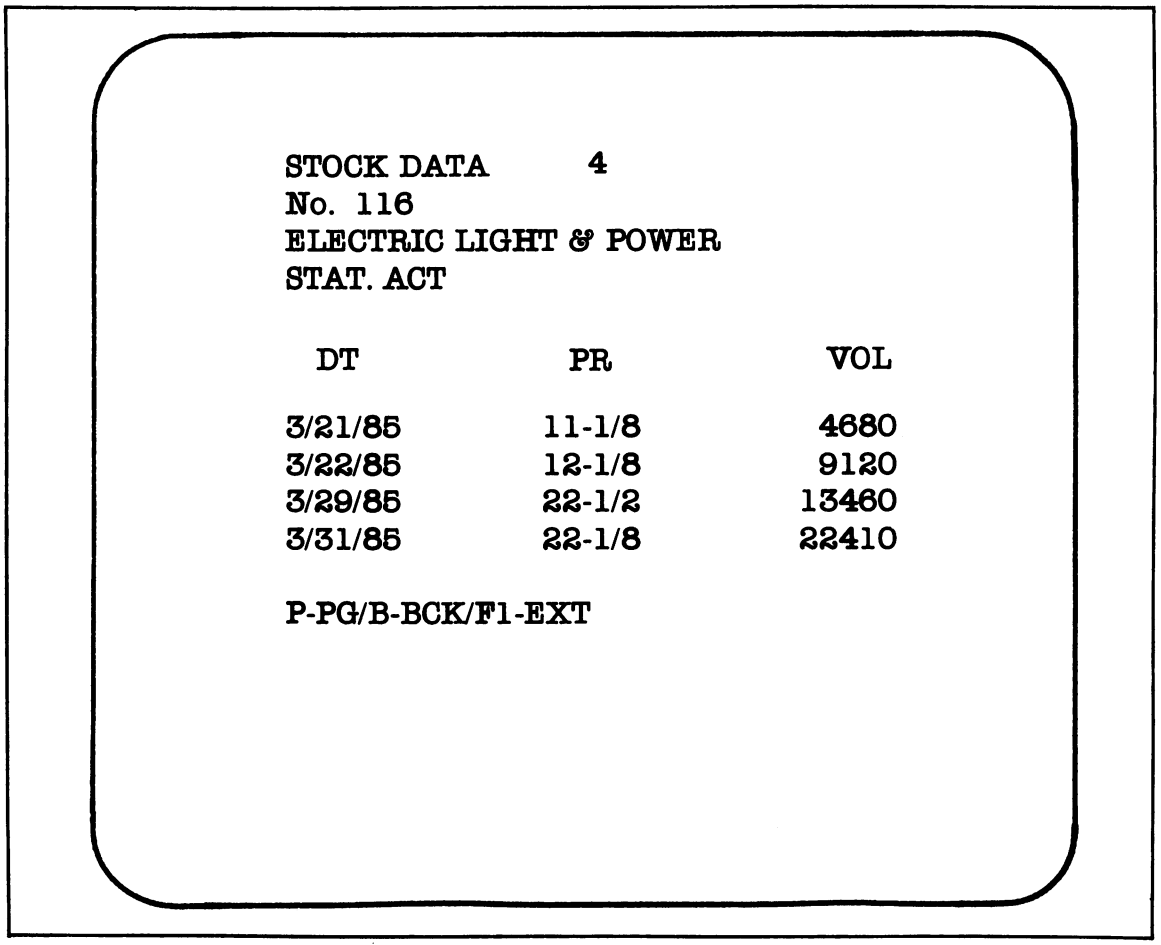


Fig. 4-21. Example of a poorly designed display screen. There are no titles; data areas overlap; headings and data appear similar, and abbreviations and prompt are ambiguous.

worst possible error would be to divide the screen up into some areas which displayed black characters on a white background, and other areas which displayed white characters on a black background. Keep the backgrounds within a narrow contrast range.

C-64 BASIC permits you to set only one background color—by POKEing the color into memory location 53281 (see Fig. 4-1). You cannot set different parts of the display to different background colors. You can, however, achieve much the same effect by printing reverse foreground colors on the screen. Figure 4-23 is the listing of a subroutine that will fill one or more rows of the screen with a specified color. The arguments of this subroutine are CHR and H1. CHR is the character code of the color the screen is to be painted (see Table 4-1). H1 is the number of rows (1-24) to paint the specified color.

Line 1610 sets the printing mode to reverse video. Line 1620 sets the character color. Lines 1630 through 1650 print H1 lines consisting of 39 spaces each. The effect of all this is to paint the screen with H1 lines of the specified color.

Note that the cursor is not positioned at the beginning of the subroutines. This must be done within program code. In addition, character color at the end of the subroutine remains as set by its

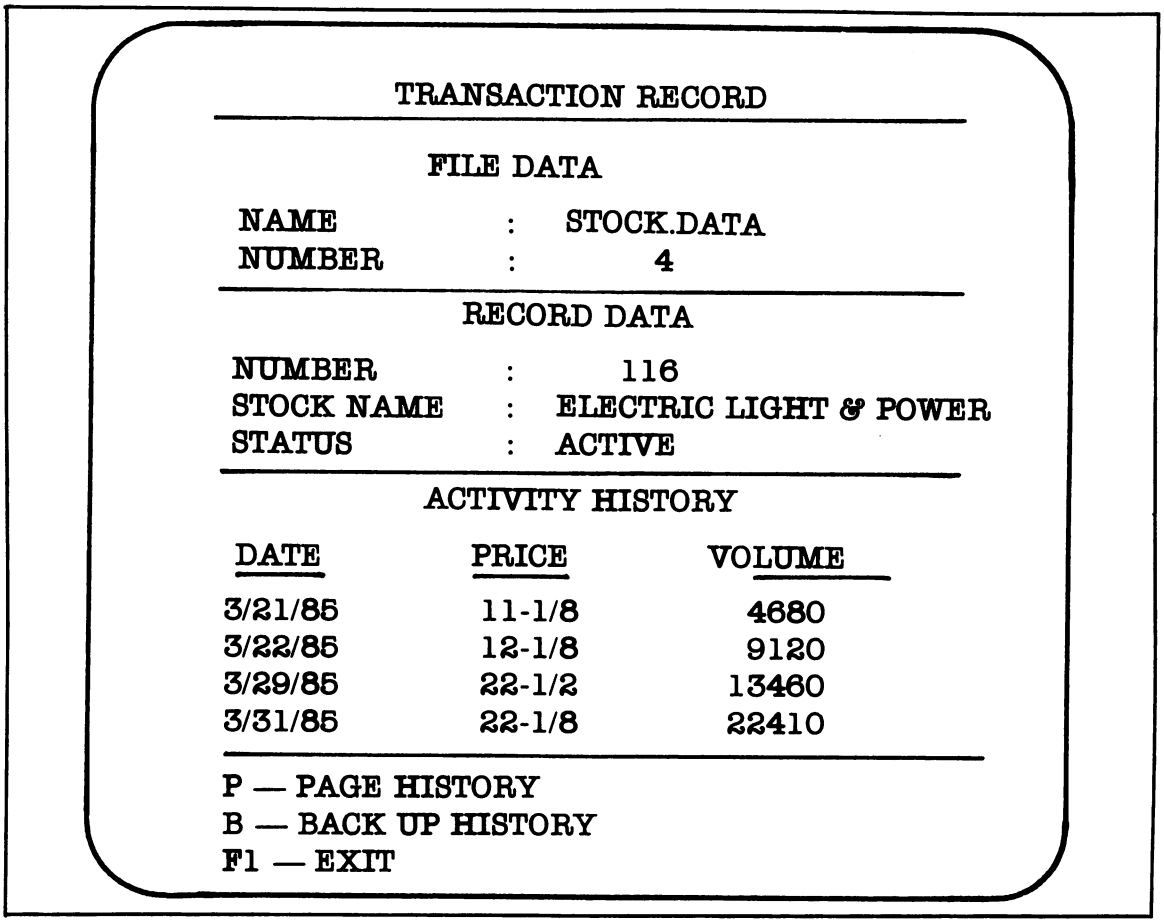


Fig. 4-22. An improved version of the display screen shown in Fig. 4-21. Titles are used; data areas are separated; full words are used instead of abbreviations, and the prompt is explicit.

argument. These two facts mean that you must position the cursor before calling the subroutine, and set the character color appropriately at its end.

To illustrate how this subroutine might be used, let us write a little program that will (1) set the screen and border initially to black, (2) create an upper 10-row area of cyan, (3) create a middle 10-row area of light red, and (4) print the words "COMMODORE 64" in yellow at the top. The code to do this is shown in Fig. 4-24.

Take a moment now to type in both the subroutine and program listing shown in Fig. 4-24. Be especially careful when you type in line 1630 to include exactly 39 spaces.

```
1600 REM--COLOR PRINTER--
1610 PRINT CHR$(CHR);
1620 FOR A=1 TO H1
1630 PRINT CHR$(18)"                                    "
1640 NEXT
1650 RETURN

READY.
```

Fig. 4-23. Subroutine to set one or more display rows to a particular character color.

```
990 GOTO 10000
1600 REM--COLOR PRINTER--
1610 PRINT CHR$(CHR);
1620 FOR A=1 TO H1
1630 PRINT CHR$(18)"
1640 NEXT
1650 RETURN
10000 REM--COLOR SETTING PROGRAM--
10010 POKE 53280,0:REM SET BORDER TO BLACK
10020 POKE 53281,0:REM SET SCREEN TO BLACK
10030 PRINT CHR$(158):REM SET CHARACTER COLOR TO YELLOW
10040 PRINT CHR$(147);:REM CLEAR SCREEN
10050 H1=10
10060 CHR=159:REM SET CHARACTER COLOR TO CYAN
10070 GOSUB 1600
10080 CHR=150:REM SET CHARACTER COLOR TO LIGHT RED
10090 GOSUB 1600
10100 PRINT
10110 PRINT CHR$(158);:REM SET CHARACTER COLOR TO YELLOW
10120 PRINT CHR$(19);:REM HOME CURSOR
10130 PRINT"COMMODORE 64"

READY.
```

Fig. 4-24. Program to set screen and character colors and to demonstrate the subroutine shown in Fig. 4-23.

Lines 10010-10030 set the screen and character colors to black and yellow, respectively.

Line 10040 clears the screen and Homes the cursor.

Lines 10050-10070 set the number of lines (H1) to 10, character color to cyan, and call the color printer subroutine. Line 10080 sets character color to light red, and 10090 calls the subroutine a second time, painting the next 10 lines of the screen.

Line 10100 is a PRINT statement that cancels reverse video mode (remove this and see what happens to your display).

Lines 10110-10130 reset character color to yellow, move cursor to Home, and print the words "COMMODORE 64."

Modify this technique, as necessary, to suit the types of screens you generate.

HOW TO DISPLAY TEXT

Text consists of words combined in sentences to communicate information. It may be used in many different ways in a program, such as:

- Written directions
- Error messages
- Warnings
- Explanations
- Descriptions
- Definitions

Here are the rules for presenting text:

- Left-justify it.
- Use uppercase and lowercase rather than all capital letters.
- Avoid word wrap—do not break a word between lines. Avoid splitting words wth hyphens, if possible.
- Avoid abbreviations and jargon.
- Use a simple, direct style, with commonplace words.
- To make extended text more readable, break it into short paragraphs.
- In giving directions or describing a procedure, list each step separately and in the order it occurs or should be performed. Avoid treating it as a long block of text or describing steps in some other order than they are performed.
- Present text in a color and against a

background field that will be easy on the eyes.

Let the user control the rate at which text is presented. For example, if you have several pages of text to present, let the user page through them by pressing the space bar, Return key, or taking some other action. Do not pace the pages with a timing loop—that is, force the user to adapt his or her reading speed to the rate at which the program presents the information.

In general, text displayed in mixed uppercase and lowercase letters is more legible and easier to read than text in all capital letters. Many programmers get into the habit of working with all uppercase, and regard it as quite natural to create screens that contain nothing but capital letters. This is similar to the old convention of filling displays with abbreviations and computer jargon. Neither of these is very communicative to the user. It is much better to employ full, natural language—avoiding abbreviations and jargon—and to use both alphabet cases available.

While using mixed uppercase and lowercase is generally good, C-64 programmers need also to recognize that the lowercase characters available to them will win no prizes for typographical design. The lowercase characters are rather crude. Mixed-case C-64 text has the appearance of something written on a chalkboard by a small child just learning the alphabet. The poor character design may, in fact, be distracting to some users. Still, it probably remains true—as research studies have shown—that mixed-case text is more readable than all uppercase.

If you do intend to use lowercase characters in text, it is best to work with your computer in the lowercase mode and to get used to the appearance of a program listing in this form. When you are in this mode, you can display both lowercase letters and their uppercase equivalents by using the Shift and letter keys together. You cannot do this in uppercase mode, since lowercase characters are no longer available.

Whether you intend to use mixed-case text or uppercase alone, it is important to take special note of four CHR$ codes:

CHR$(14)—Shifts the display to lowercase mode.

CHR$(142)—Shifts to uppercase mode.

CHR$(8)—Disables the shifted Commodore key and locks the display in whatever case mode it is in.

CHR$(9)—Enables the shifted Commodore key and permits case changes via the keyboard.

These CHR$ codes are important because, by PRINTing them within a program, you can change the case mode or lock or unlock the case mode. For example, to put the display in lowercase mode, include in your program the statement PRINT CHR$(14). To return to uppercase, PRINT CHR$(142).

Once you have selected a case mode, you do not want the user to be able to change it from the keyboard. Therefore, early in the program include the statement PRINT CHR$(8). This turns off the shifted Commodore key and prevents the user from making case changes from the keyboard, but still lets you make case changes within the program by using the appropriate CHR$ codes. When the program ends, reactivate the shifted Commodore key by PRINTing a CHR$(9).

HOW TO DISPLAY NUMBERS

As just noted, the convention for presenting text is to *left-justify* it, or align the text against a common left margin. A different convention is followed for presenting numbers. Numbers are aligned on their decimal point (Fig. 4-25). In addition, they are usually displayed with the same number of significant digits (numbers to the right of the decimal point). Some BASICs have formatting statements that make printing numbers this way easy. C-64 BASIC does not, but you can write a BASIC subroutine that will do the job (such a subroutine is described later in this chapter).

Aligning numbers on the decimal point is called *$Formatting*, since the technique is widely used for printing columns of monetary values. For example, bills are formatted this way, and so are checkbook reconciliation statements that are sent

out by banks. The convention is followed for displaying columns of most numerical quantities.

Displaying numbers this way makes them easier to read. Compare, for example, Fig. 4-25 ($ formatted) and 4-26 (unformatted) and ask yourself these questions:

- Which looks neater?
- Which will help you find the largest and smallest number most quickly?

In addition to alignment on the decimal point, a set of related numbers is usually displayed in a column, beneath a heading that tells what the numbers represent. Never display a set of numbers like text—in rows that the reader must scan (Fig. 4-27).

If you are displaying a single number, or a series of unrelated numbers, place a header to the left of each number and separate it from the number with a colon, like this:

PRICE: $105.16

In short, there are two conventions for displaying numbers:

- List of numbers—In a column, aligned on the decimal point, with a header at the top.
- Single number—Heading at the left, colon in the middle, number on the right.

The remainder of this section deals mainly with the first convention, culminating in a subroutine for doing $ formatting; however, a few preliminaries must be covered first.

Commodore Abhors Naked Numbers

Ever heard the statement that "Nature abhors a vacuum?" Well, the C-64 feels the same way about "naked" numbers, that is, numbers with no spaces around them. To illustrate, consider the number 1.

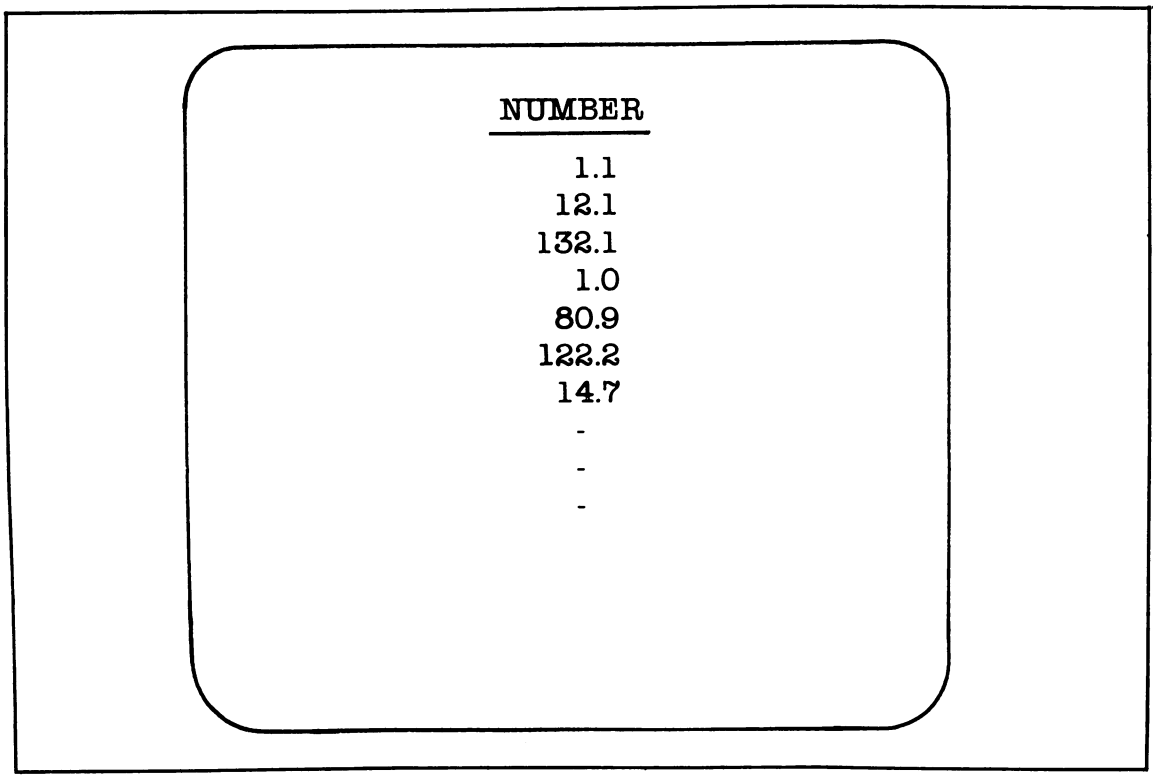


Fig. 4-25. A column of numbers aligned on the decimal point. This is the proper way to present numerical information.

How many characters are in this? Do you think that you know? Try a little experiment. Type this statement into your C-64, press Return, and see what answer you get.

```
PRINT LEN(STR$(1))
```

The STR$ statement converts the number 1 to string form and the LEN statement measures the length of the string. The answer you get back is 2. Apparently, the C-64 thinks that there are two characters, not one.

Well, actually, it does not think anything. It knows that there are two characters because it added a space in front of the 1 when it converted it to a string. It does this every time it converts a positive number to string form. Now type this into your C-64 and press Return:

```
PRINT LEN(STR$(-1))
```

The answer this time is also 2. This may suggest to you why BASIC added an extra space in front of the positive 1—so that positive and negative numbers contain the same number of characters.

BASIC's intentions may be good, but the unsolicited help it offers sometimes produces undesirable complications. For example, suppose that you want to convert a number to a string (take my word for now that this is often a useful thing to do) and then to PRINT the string at the left edge of your display. Sounds like a simple thing to do. Try typing this in at your C-64:

```
N$=STR$(-1)
PRINT N$
```

BASIC prints the number 1, all right, but indents it one space to the right. It will not do this with negative numbers, as you can prove to yourself by typing in this:

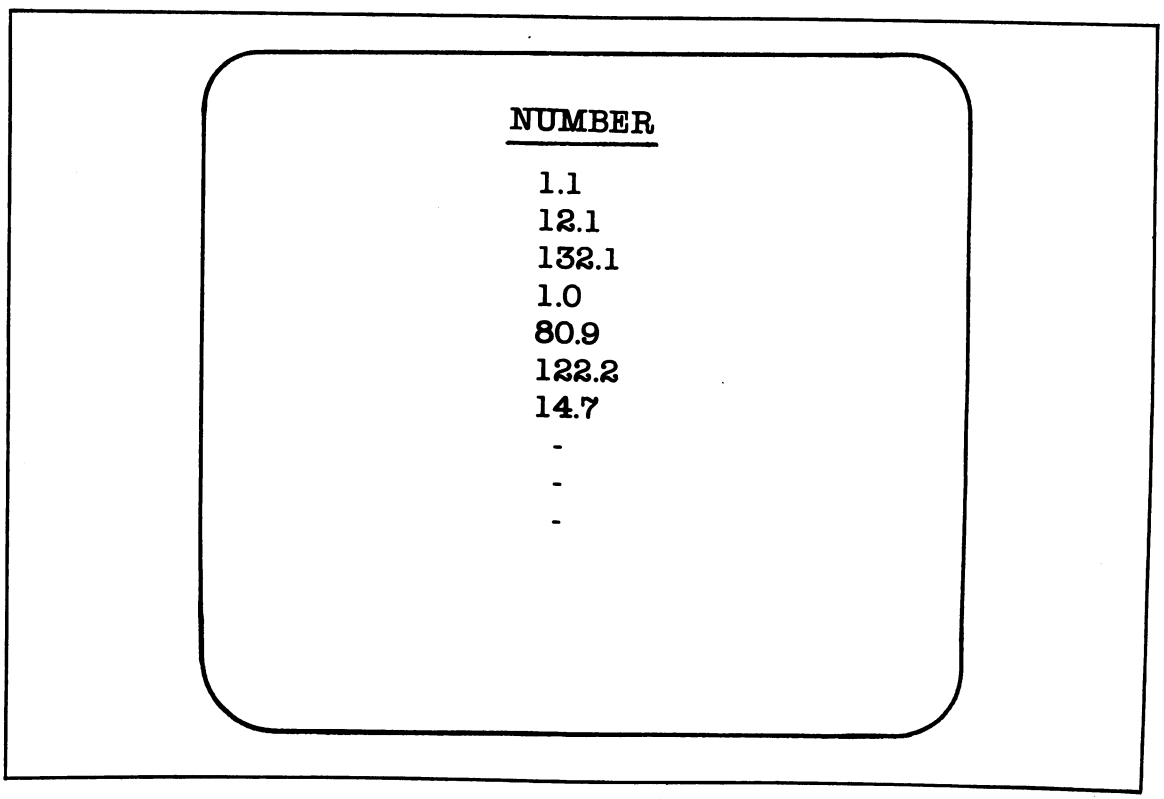


Fig. 4-26. Numbers left justified: This is a poor way to present numerical information.

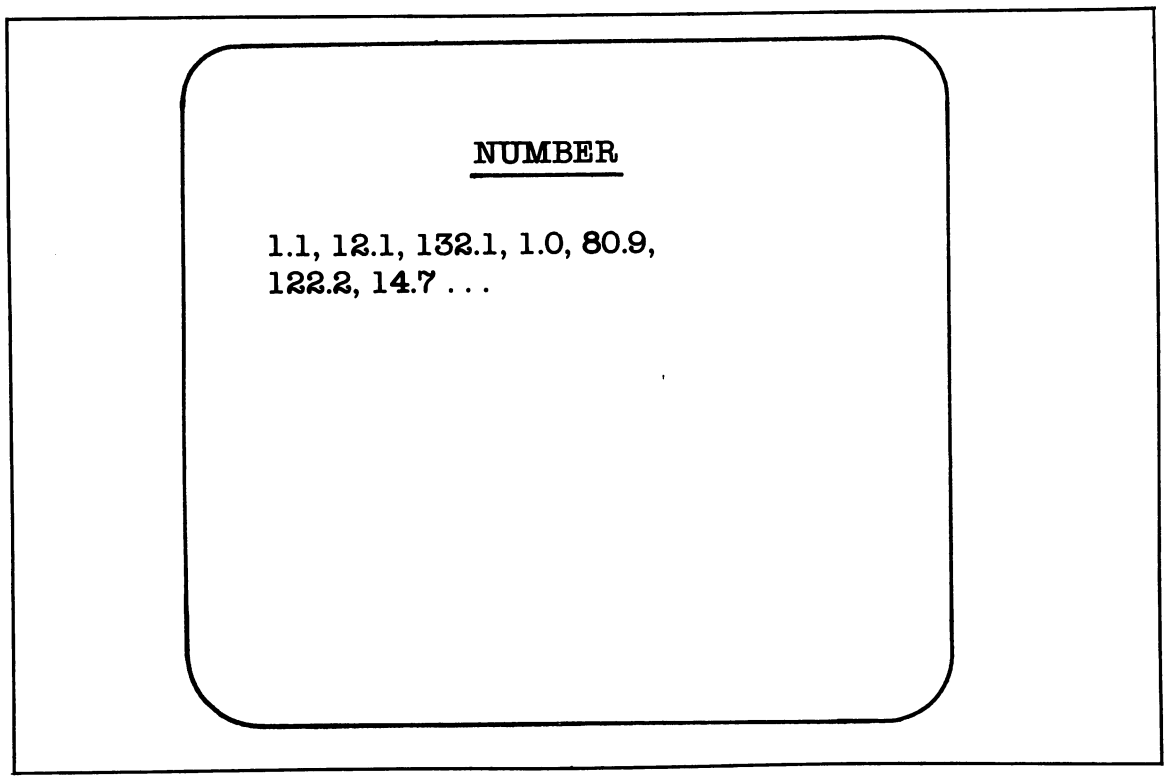


Fig. 4-27. Numbers printed like text, in rows. This is a poor way to present numerical information.

```
N$=STR$(-1)
PRINT N$
```

This prints at the margin, where it belongs.

The main problem produced by the extra space is that you cannot trust a number to print on your display where you tell it to. If the number is positive, BASIC adds an extra space, but not if it is negative. You do not always know whether the numbers you intend to display will be positive or negative, and this means that, unless you modify BASIC's unequal treatment of positive and negative numbers, you cannot properly format a display that contains numbers.

The solution to the problem is to put any numbers through a filter that removes the leading blank if the number is positive. Figure 4-28 is the listing of a subroutine that converts any number to its string equivalent without adding an extra space in front of positive numbers. This subroutine has one argument, N, the number to be converted. It returns the string equivalent of N in the string variable N$.

Line 1810 converts N to the string N$.

Line 1820 calculates the length of N$ and assigns this value to L.

Line 1830 tests N to see if it is greater than or equal to 0. If so, it subtracts 1 from L, the computed length of N$.

Line 1840 takes a substring of N$ of length L. If N is positive, this removes the first character—a blank—from N$. If the number is negative, the substring is the complete string.

How to Set the Number of Decimal Places

Many hand-held calculators have a feature that permits you to set the number of decimal places to be displayed. You know, you set a switch to 2 if you want two significant digits, to 4 if you want four, or to 0 if you are interested in rounding things off to

```
1800 REM--NUMBER TO STRING CONVERTER--
1810 N$=STR$(N)
1820 L=LEN(N$)
1830 IF N>=0 THEN L=L-1
1840 N$=RIGHT$(N$,L)
1850 RETURN
```

Fig. 4-29. A subroutine to convert a number to its equivalent string form, without adding a leading space when the number is greater than or equal to zero.

whole numbers. The C-64, like most microcomputers, does not have this feature. If you start performing mathematical calculations that involve decimals, the number of decimal places displayed sometimes increases to preposterous lengths. Often you wind up with nine decimal places when all you want is one or two.

For example, sit down at your keyboard and type in the following:

PRINT 1/6

Your computer displays .166666667. It is easy enough to round this off in your head, but it would be nice to have the computer do it for you. You would probably be happy to know this value to two or three decimal places and, if it was produced by a calculation, might not trust a calculation that had any more places than this anyway.

The simplest way to round numbers is with a user-defined function. The following is the general form of a function that can be used for this purpose:

FN ROUND (X) =INT (X * 10 * Number of Places + .5)/10 * Number of Places

You can tailor this general function to suit a specific purpose by substituting a number for the parameter Number of Places. To illustrate, Fig. 4-29 shows three definitions of specific functions for rounding X.

Line 410 contains the definition of FN C1 (X), which rounds X to one place.

Line 420 defines FN C2 (X), which rounds to two places.

Line 430 defines FN C3 (X), which rounds to three places.

Due to floating point errors, these functions will sometimes produce slightly inaccurate results when rounding numbers ending in exactly .5—such as 1.5, 2.5, etc. You can correct the functions for this by substituting .500001 for .5 in the function definitions.

In the improbable event that you have not used a user-defined function in a program before, here is a quick lesson. First, define the function early in the program. When you want to use it during the program, simply use it like any other BASIC statement. For example, to have the function round the value of your variable Y1 to three places, insert a statement setting Y1=FN C3(Y1).

Try these functions out for yourself. Type in the listing in Fig. 4-29 and add the following lines:

10000 INPUT "X:";X

```
400 REM--DEFINE FUNCTIONS--
410 DEF FN C1(X)=INT(X*10+.5)/10:REM ROUND TO 1 PLACE
420 DEF FN C2(X)=INT(X*100+.5)/100:REM ROUND TO 2 PLACES
430 DEF FN C3(X)=INT(X*1000+.5)/1000:REM ROUND TO 3 PLACES
```

Fig. 4-29. User-defined functions to round the variable X to 1, 2, or 3 decimal places.

```
10010  PRINT FN C1 (X)
10020  PRINT FN C2 (X)
10030  PRINT FN C3 (X)
```

RUN this little program, enter the value of X, and watch the rounded values be displayed.

$ Formatter

Figure 4-30 contains the listing of a $ Formatting subroutine. The arguments of this subroutine are N, N0, and T. N is the number to format. N0 is the number of decimal places to round the number to. T is the tab value at which the decimal point should appear.

To illustrate the subroutine's use, suppose that you calculated the value of variable X1 during a program and wanted to round and display this variable with three significant digits and with the decimal point appearing at tab position 32. You could round and display this number by inserting the following statements into your program:

```
10010  N=X1
10020  N0=3
10030  T=32
10040  GOSUB 2000
```

The arguments are assigned in lines 10010-10030, and the subroutine is called in line 10040. Simple enough.

Here is how the subroutine works. Line 2010 rounds the value of N to the number of places specified by its argument N0. You may recognize the formula on this line as identical to the general formula described in the discussion of user-defined functions for rounding numbers.

Line 2020 converts N to a string.

Line 2030 computes L, the length of the string. For example, converting 1234 to a string will yield a length (L) of 5 (number of characters plus the added space at the front of the string).

The next few lines locate the position of the decimal point, if present, within the number. To ensure that the decimal point is later printed at the correct tab position, the first character in the number must be printed starting as far left of the decimal point as there are spaces between the first character and the decimal point. For example, if you desire to print the number 123.45 with the decimal point at tab position 23, then the first character must be printed starting at tab position 20. If the number does not contain a decimal point, then it must be printed its entire length to the left of the designated tab position. This distance to the left of the position of the decimal point is referred to as the "tab offset."

Line 2040 initializes the tab offset to L+1, the length of the number plus 1. The 1 is added in case the number does not contain a decimal point; this has the effect of defining the location of the decimal point to the right of the number.

Lines 2060-2070 search through N$, from left to right, looking for a decimal point. If one is found, its location is defined by setting T0=A. For exam-

```
2000 REM--$ FORMATTER--
2010 N=INT(N*10↑N0+.5)/10↑N0:REM SET DECIMAL PLACES
2020 N$=STR$(N)
2030 L=LEN(N$)
2040 T0=L+1:REM INITIALIZE TAB OFFSET
2050 FOR A=1 TO L
2060 IF MID$(N$,A,1)="."THEN T0=A:REM LOCATE DECIMAL POINT
2070 NEXT
2080 PRINT TAB(T-T0)N$;
2090 RETURN
```

Fig.4-30. $ Formatter subroutine for printing a number on the video display with a specified number of decimal places and its decimal point at a specified column number.

ple, if the decimal point is the fourth character in N$, then T0=4. If no decimal point is located, then the number is an integer and it retains the initial T0 value set in line 2040.

Line 2080 prints N$ at tab position T-T0.

Try this subroutine out for yourself. Type it in. Add the following lines so that you can enter values and see what results you get on your screen:

```
100  INPUT "N:";N
110  INPUT "NO:";NO
120  INPUT "T:";T
130  GOSUB 2000
140  PRINT
150  GOTO 100
```

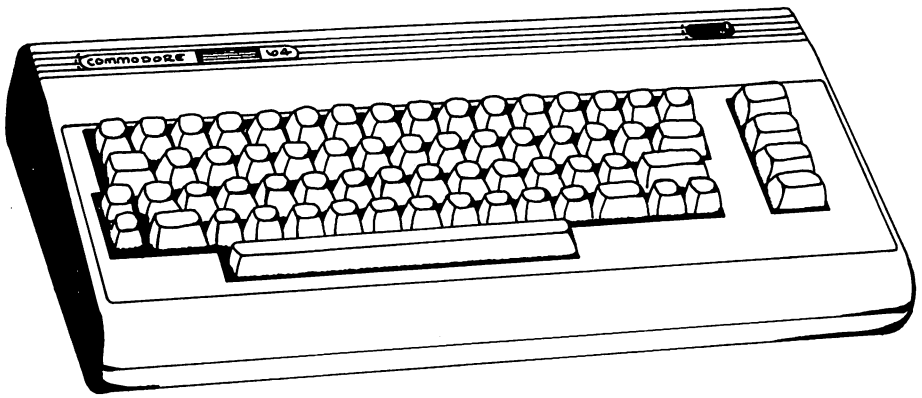
This version of the subroutine uses the TAB statement for cursor positioning. This works fine if you always print numbers from left to right on your screen and numbers never overlap. If you want greater flexibility in using the subroutine, modify line 2080 so that it uses a SYScall, as described earlier in this chapter. That is, change line 2080 to read:

```
2080  SYS C0,0,V,(T-T0):PRINT N$;
```

If you make this change, note that the number will print one space to the left of where it would print using the TAB statement, since the SYScall measures the cursor's horizontal position differently. That is, TAB measures columns from 0-39, but the SYScall measures them from 1-40.

Note that line 2090 ends with a semicolon so that no line feed is issued when this line is executed, thereby permitting you to format several numbers on the same print line. If the last thing you PRINT on a line is a formatted number, however, you must issue a separate PRINT statement to cause a line feed.

Chapter 5



Data Entry, Error-Testing, and Validation

One of the first things a novice BASIC programmer learns is the INPUT statement. It is easy to use, and most new programmers master it within their first few hours of programming. Only later do they learn that there is much more to collecting data from program users than this simple statement. Not only are there other ways to intercept keystrokes—the GET statement, for example—but many complications arise whenever a program must handle data entries.

To begin with, users do not always know what to enter. The prompts the programmer has written may be unclear, or the user may not have read the user's guide. In their confusion or ignorance, program users then type in information that the programmer had never in the wildest flights of imagination expected. The user will press the Return key with no typed entry, enter numbers where letters were expected and vice versa, invent creative syntaxes for typing in dates, times, and telephone numbers, and perform other tricks to confound the programmer's neat and orderly view of the world. Well, when these things happen, it does not take the programmer long to wise up. He or she then concludes that something more than the INPUT statement is required. But what?

Good prompting, for one thing. With good prompts, the user is less likely to make incorrect entries. This by itself, however, is not enough. Since errors will still occur some of the time, the programmer realizes that all user entries must be tested before they are accepted by the program. No programmer can read the minds of all program users, or anticipate every possible entry error, any more than he or she can write code to test for every error. Error-testing, it suddenly becomes clear, is more complex and difficult by far than using BASIC statements to convert these keystrokes into variables in memory.

Once the programmer finds a practical way to handle error-testing, the next problem comes to light—after making entries, the user reconsiders or recognizes an error, and wants to change the entry. Is it too late? Should it be too late? Should it be possible to start the entry process all over again?

This chapter attempts to provide reasonable

answers to these and related data entry questions. I say "reasonable" because the truth is that it is impossible to write prompts that everyone understands, test for and protect against every possible entry error, or satisfy the user's every whim. It is possible, however, to do these things reasonably well—which is about all that should be expected from any reasonably sane, reasonably compulsive programmer.

The chapter begins with a discussion of data-entry statements. Next, the various steps in the data-entry process—prompting, collecting keystrokes, error-testing, verification—are discussed, and programming techniques, examples, and relevant subroutines are described. Tips and hints are then provided for integrating data entry routines with on-screen displays.

DATA-ENTRY STATEMENTS

Your C-64 has four statements that may be used for taking data entries from the keyboard:

- INPUT
- INPUT#
- GET
- GET#

The INPUT statement is the most commonly used of these. The INPUT# statement is normally used for reading in data from files, but it can also read the keyboard. It has some additional, desirable features that the INPUT statement lacks. GET reads one character from the keyboard, and does not require the user to press the Return key afterwards. GET# is to GET what INPUT# is to INPUT—the version of GET used to read data files. Like INPUT#, GET# can also be read in the keyboard; however, it is the functional equivalent of GET, and offers no additional features.

The INPUT, INPUT#, and GET statements have different characteristics. This section discusses each of them in turn. Because of the similarity of GET and GET#, the GET# statement is not covered here. Each of these three statements has slightly different characteristics. Let us consider each one separately.

INPUT Statement

The main characteristics of the INPUT statement are as follows:

- It presents a question mark and flashing cursor on the screen, which attract attention.
- It can be used for collecting long entries.
- The user can modify the entry before it is assigned to the statement's variable. The entry is verified by pressing the Return key.
- The user can disrupt the appearance of the display by making inappropriate entries. For example, if a real variable follows the INPUT statement (for example, INPUT A) and the user presses the Return key with no entry, the message "REDO FROM START" appears on the screen and the screen scrolls. If a string variable follows the INPUT statement, repeated pressing of the Return key causes the screen to scroll. If the user types in too long an entry, the screen scrolls.
- If a string variable follows the INPUT statement, and the user presses the Return key with no typed entry, BASIC does not convert the variable to the null string, but leaves it with its existing assignment. To illustrate, type in and RUN this program. Press the Return key with no entry when the prompt appears.

```
10  A$="OLD ASSIGNMENT"
20  INPUT "ENTRY:";A$
30  PRINT A$
```

The value of A$ printed is OLD ASSIGNMENT. The solution to this problem is to assign all string data-entry variables to the null string before using them in INPUT statements. That is, modify line 20 to read.

```
20  A$="":INPUT"ENTRY:";A$
```

INPUT# Statement

Most C-64 programmers are unaware that it is

possible to use the INPUT# statement to collect data from the keyboard. This statement is most commonly used to read in data from data files. Before it works, the program must OPEN a file and specify a device from which data are to be read in. The OPEN statement has this syntax:

OPEN File Number, Device Number.

For reading or writing a data file, *file number* is some number between 2 and 14, and *device number* is 8 (drive #1) or 9 (drive #2).

The keyboard has a device number (1), so you can OPEN a file and read in data from it. To illustrate, let us set file number to 1 and write a little program to OPEN and read the keyboard. Here it is:

```
10 OPEN 1,0
20 PRINT"ENTER DATA:";
30 INPUT #1,A$
40 PRINT
50 CLOSE 1
```

If you try this program out, you discover that the INPUT# statement works much like the INPUT statement except for the following differences:

- No question mark prints on the screen, although there is a flashing cursor.
- Pressing the Return key with no typed entry has no effect.

In other respects, the two forms of INPUT work identically. Using INPUT# requires additional code to do the following:

- OPEN and CLOSE the keyboard for input.
- PRINT the prompt—INPUT# does not permit an embedded prompt string.
- PRINT a carriage return after the entry has been taken (line 40)—INPUT# does not do this automatically.

Since INPUT# does not respond to a null entry, it cannot inadvertently pick up its variable's previously assigned value. To illustrate, add this line to the preceding listing:

```
5 A$="OLD ASSIGNMENT"
```

Now RUN the program. Press the Return key with no typed entry.

As you discover, the cursor flashes contentedly away, and ignores the Return key press. The only way to move on is to type in a more substantial entry—one or more characters. Type in a letter and press Return and the display responds. You can still type in too many characters, however; so there is nothing to prevent you from making your display scroll, if you or some mischief-minded user wants to do it. If the possibility of scrolling is a major concern, then there is a way to use the GET statement (see the discussion of the "Deluxe GET", later in this chapter) to prevent the user from entering too many keystrokes.

Using the INPUT# in a program requires slightly more code than using INPUT, but the extra effort is well worth it in terms of increased error protection and more professional-appearing data-entry prompts. In fact, for reasons that will be apparent presently, INPUT# is the preferred method to collect keystrokes in the majority of programs.

GET Statement

The main characteristics of the GET statement are as follows:

- It does not produce visual cuing on the screen—no flashing cursor or question mark. (A flashing cursor can, however, be produced by POKEing the value 0 into memory location 204.)
- It collects one character.
- It does not "echo" the entry on the screen—the typed entry is taken by the computer but not displayed.
- It does not permit the user to verify the entry. Once typed in, the entry is assigned to its variable.

- Since GET collects only one character, there is little chance of disrupting the appearance of the display through improper data entry. (Attempting to get a real variable produces a SYNTAX ERROR message, but doing this is a mistake anyway.)
- If the variable following the GET statement has already been assigned, and the user presses the Return key with no entry, the variable is assigned to the null string. To illustrate, here is the GET version of the short data-entry program used to illustrate, a problem with the INPUT statement:

```
10  A$="B"
20  GET A$:IF A$="" GOTO 20
30  PRINT A$
```

Type in and RUN the program and simply press the Return key. The screen remains blank, indicating that the value of A$ is the null string, not the character "B" assigned on line 10.

GET is a different kind of data entry statement than INPUT or INPUT#. It takes one keystroke at a time, and does not require the user to press the Return key to verify the entry. It would seem that the naked GET is useless in most programs, since it does not have a flashing cursor nor does it echo entries to the screen. Let us not jump to this conclusion yet, however.

Actually, there are three possible ways to think about GET—sort of like buying a car with different options. Let us consider the three models:

Naked or "Economy" GET. This is GET in its simplest form. Here is an example:

```
10  PRINT "DO YOU ACCEPT THE RESULT?
(Y/N):";
20  GET A$:IF A$="" GOTO 20
30  If A$< >"Y" AND A$ < > "N" GOTO 20
```

This little program is used to verify a result that was previously displayed to the program user. If the user accepts the result, he or she types in Y. If not, an N is typed in. In this program, line 10 displays the prompt, 20 contains the GET loop, and 30 runs an error test to make sure that the routine cannot be exited until either a Y or N is typed in. In a simple situation like this, no flashing cursor is placed on the screen, and the typed entry is not displayed. The routine would be better if they were, but in such a simple application, these additional features are not critical.

Improved GET. The Improved GET has two features not included in the Economy GET—a flashing cursor and display of the entry. We can obtain these features by adding two lines to the code. Here is the modified listing:

```
10  PRINT "DO YOU ACCEPT THE RESULT?
(Y/N):";
15  POKE 204,0:REM FLASH CURSOR
20  GET A$: IF A$="" GOTO 20
30  IF A$< >"Y" AND A$ < >"N" GOTO
40  PRINT A$:½EM DISPLAY KEYSTROKE
```

Deluxe or "Luxury" GET. The Deluxe GET does everything that the Improved GET does, and also allows multiple-keystroke entries. It allows entry of commas (INPUT and INPUT# do not), permits you to filter what the user types in by ignoring certain characters, and limits the entry to a minimum and maximum length. This GET is much more powerful than the INPUT or INPUT# statements, and would be used in place of them in certain types of programs.

In the following form, it can be used to create complete data entry screens, such as Fig. 5-1. Such a screen contains several complete data entry fields, and the user types in the entries for one field at a time. After completing field one, the cursor moves to field two, three, and so on. In such screens, it is important to have complete control of what the user types in and to prevent any keystrokes from disrupting the screen. Without this control, a few unfortunate keystrokes can turn the screen into an unreadable mess.

The code for the Deluxe GET is a bit involved, and the best way to describe how it works is with a flowchart that shows its logic and the functions that must be performed. After going through the chart, we will look at the actual code.

In using GET to collect an entry that is more

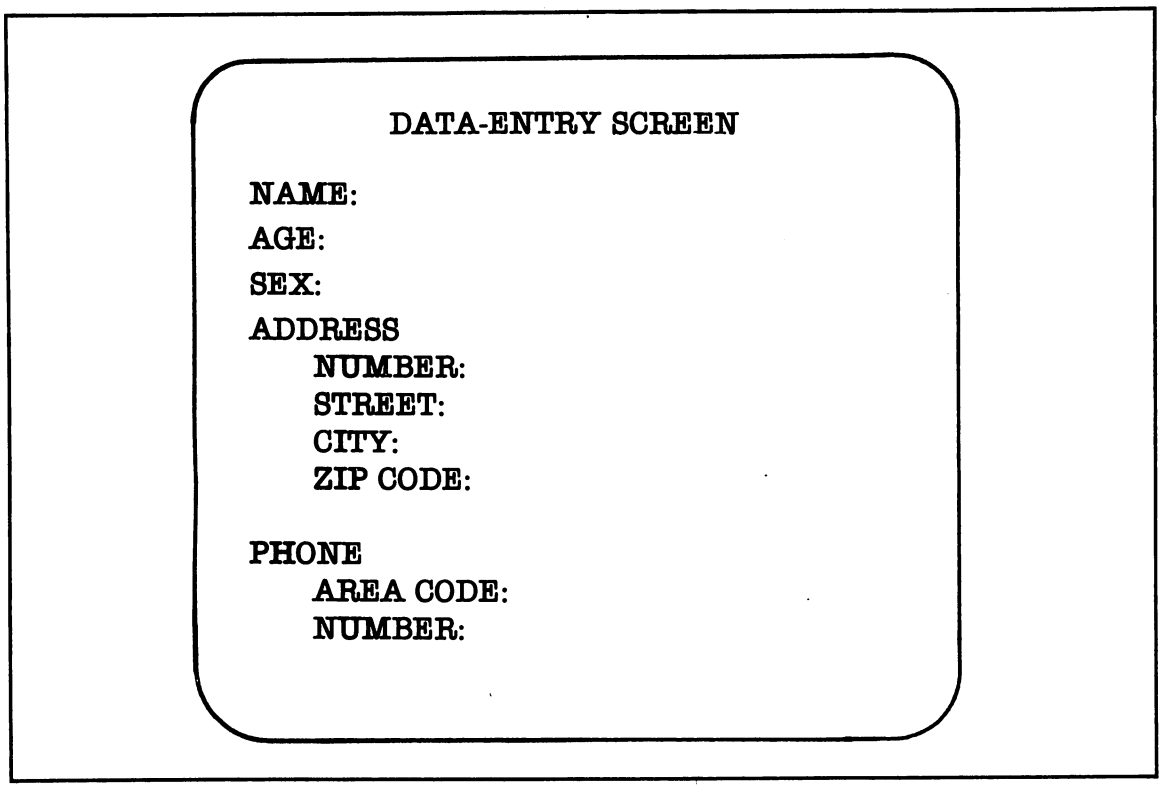


Fig. 5-1. Data-entry screen. All prompts are printed on the screen, and the cursor moves from field to field as the user types in data.

than one character long, the code must still collect one character at a time. It then concatenates the characters together into a longer string. Not all characters can be treated equally, however. A Return key press (CHR$(13)) signals the end of data entry. Pressing the DELETE or BACKSPACE key means that one character must be removed from the current entry. Characters can also be selectively ignored.

Figure 5-2 is a flowchart showing the logic of the Deluxe GET. Each of the steps in the flowchart has a number. The number on the right of each step is the line number in the corresponding BASIC listing shown in Fig. 5-3. In both the flowchart and the listing, the entire entry is represented as A$ and the single character being GETed by B$.

Step 1 positions the cursor to a particular part of the screen, and Step 2 PRINTs the prompt. Note that no upward arrows return to either of these steps. This means that it is possible to fill a complete screen with prompts, and then go through Steps 3 through 12 of each data entry sequence separately.

Step 3 positions the cursor again—this time at the end of the prompt, at the beginning of the data-entry field. Step 4 then clears the data-entry field by erasing whatever is in it. Step 5 prints whatever has been typed in so far. These three steps reprint the entire current entry each time a key is pressed. The main reason for doing this is to handle deletions or backspaces. Without rewriting, all previous entries would remain on the screen (the DELETE key does not work properly in a GET loop).

Step 6 prints a flashing cursor on the screen.

Step 7 GETs one keystroke.

Step 8A tests whether the keystroke is a Return key press. If so, it tests in Step 8B whether the length of the current entry (A$) is less than the

minimum allowable length (Lmin). Lmin can be set to any value less than or equal to Lmax (maximum length), including zero. It is usually desirable to limit minimum length to at least one character, however. If the minimum length test is failed, then control jumps back to Step 7 so that another character can be GETed. If it is passed, then control exits the routine.

Step 9A tests for a DELETE key (CHR$(20)) or BACKSPACE(CHR$(157)) key press. If one occurred, it tests in Step 9B whether the length of A$ is greater than zero, and, if this test passes, in Step 9C it removes the rightmost character from A$ before returning to Step 3 to reprint A$.

Step 10 tests whether A$ is equal to Lmax. If so, control returns to Step 7—if A$ has reached the length of Lmax, control will continue to recycle to Step 7 until the Return key is pressed or a character is deleted from A$.

Step 11 is the keyboard filter. It checks whether the ASCII values of the keystrokes meet acceptable criteria—comma, number, letter, or space bar. If the test is failed, control returns to Step 7—this has the effect of ignoring keystrokes that fail the test.

Step 12 concatenates the single character B$ with A$, the string comprising the entry so far.

Now let us consider the code that performs these wonders. Figure 5-3 contains the listing of the Deluxe GET routine whose logic was just described. Note that the program requires the assembly-language subroutines. These subroutines are needed in the present program to position the cursor and to clear the prompt line. It is possible to perform these two functions with BASIC code, but not as easily as with the assembly-language subroutines.

The data-entry code is contained in lines 10000 through 10310. A rather long listing to do what an INPUT statement could do, you might be thinking, but recognize that this code does considerably more.

Lines 10010 and 10020 set L1 (minimum length of A$) and L2 (maximum length). Line 10030 initializes the string A$. Line 10010 clears the screen.

Line 10050 uses a SYScall to position the cursor to row 12, column 1.

Line 10060 PRINTs the data-entry prompt. This prompt indicates that the maximum length of the name is 12 characters (the value of L2).

Line 10070 uses a SYScall to clear the line to the right of row 12, column 23. Line 10080 then locates the cursor at the same row and column. Line 10090 then prints A$, the entry so far.

Line 10100 displays the flashing cursor.

Line 10110 is a GET loop that collects one keystroke.

Line 10130 tests whether the Return key was pressed. If not, control jumps to line 10170. Lines 10140 and 10150 are executed only if the Return key was pressed. Line 10140 tests whether A$ satisfies the minimum length requirement. If not, control is sent back to line 10110 to GET another character. In this listing, line 10150 contains a PRINT statement (to provide a carriage return following the PRINTing of A$;) and then END. In an actual program, END would be replaced by a GOTO directing control to the next step in the data-entry sequence.

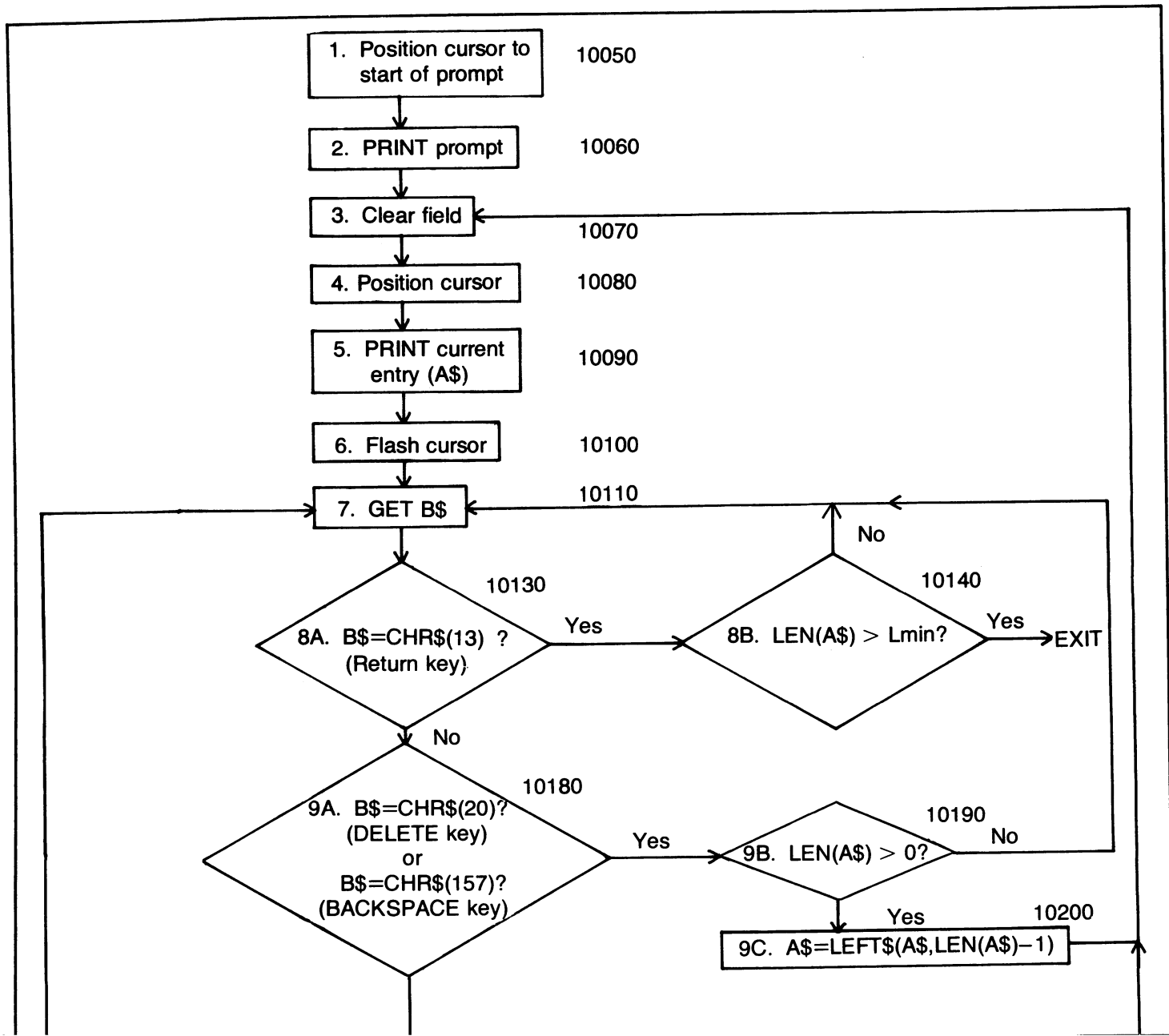
Line 10180 tests whether the DELETE or BACKSPACE keys were pressed. If so, line 10190 is executed to test whether the length of A$ is zero characters. If so, control is directed to line 10110 to GET another character. If not, then line 10200 removes the rightmost character from A$, and control is sent to line 10070, where the former A$ is erased, and the new A$ is rewritten in its place.

Line 10230 performs the maximum length test. As long as the length of A$ equals L2, control will continue to recycle to line 10110.

Lines 10240-10290 comprise the filter. These lines test whether the key press B$ is an acceptable character. If so, control jumps to line 10300, where B$ is concatenated with A$. If none of the tests succeeds, line 10290 sends control back to line 10110 to GET another character. After concatenation, control is sent to line 10070 to rewrite the new version of A$.

Comparing the Data-Entry Methods

Which data-entry method—INPUT, INPUT#,


1. Position cursor to start of prompt
10050
2. PRINT prompt
10060
3. Clear field
10070
4. Position cursor
10080
5. PRINT current entry (A$)
10090
6. Flash cursor
10100
7. GET B$
10110
8A. B$=CHR$(13) ? (Return key)
10130
Yes
8B. LEN(A$) > Lmin?
10140
Yes
EXIT
No
No
9A. B$=CHR$(20)? (DELETE key) or B$=CHR$(157)? (BACKSPACE key)
10180
Yes
9B. LEN(A$) > 0?
10190
No
Yes
9C. A$=LEFT$(A$,LEN(A$)-1)
10200

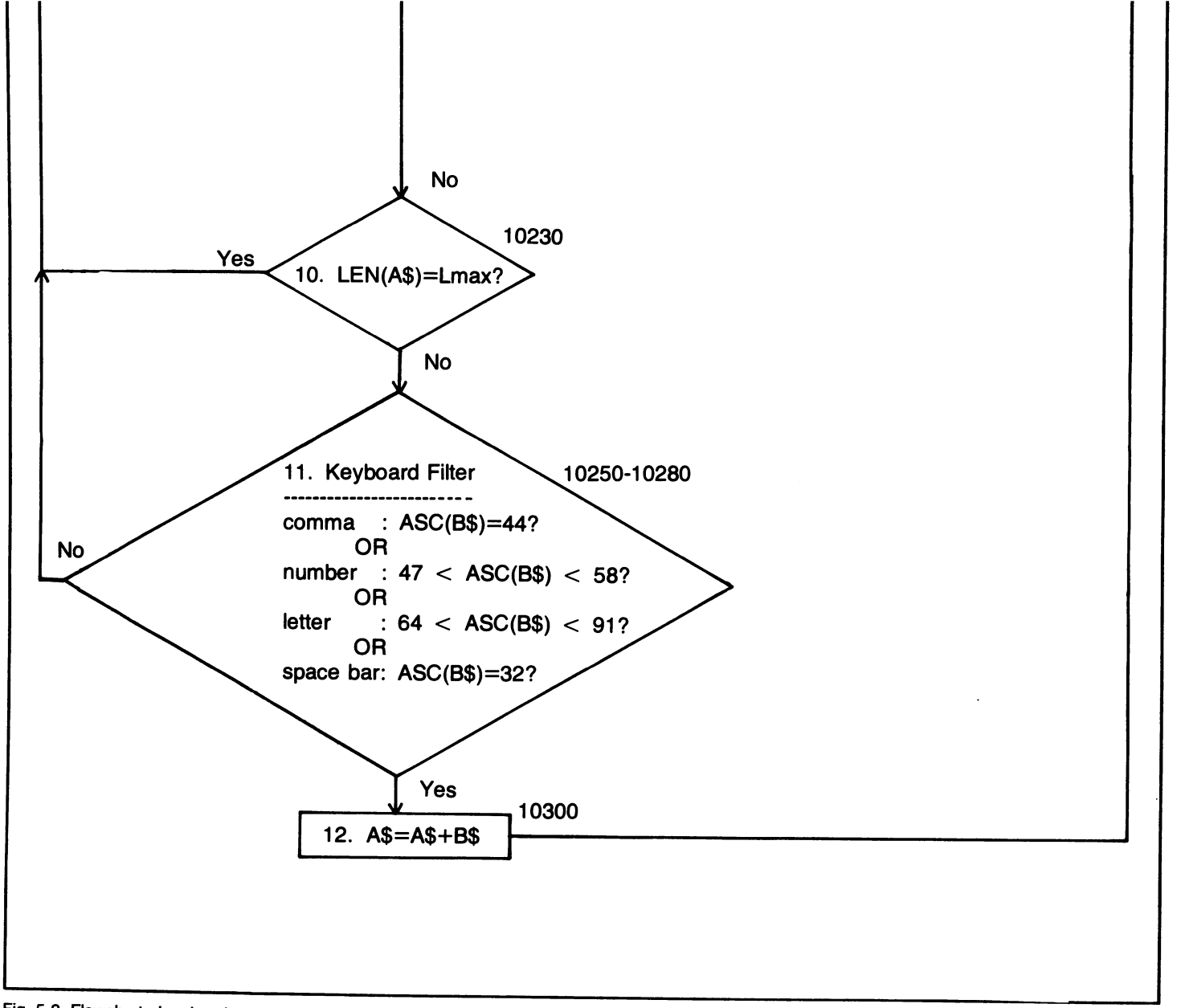


Fig. 5-2. Flowchart showing the logic of "Deluxe GET" data-entry routine.

```
10 REM ********************************
20 REM * NOTE: THIS PROGRAM REQUIRES  *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE *
40 REM * LOADER (FIGURE 4-6)          *
50 REM ********************************
10000 REM--GET ROUTINE FOR EXTENDED INPUT--
10010 L1=1:REM SET MINIMUM LENGTH
10020 L2=12:REM SET MAXIMUM LENGTH
10030 A$="":REM INITIALIZE DATA ENTRY
10040 PRINT CHR$(147);:REM CLEAR SCREEN
10050 SYS C0,0,12,1:REM POSITION PROMPT
10060 PRINT"NAME (UP TO 12 CHAR): ";
10070 SYS C0,1,12,23:REM CLEAR LINE TO RIGHT OF CURSOR
10080 SYS C0,0,12,23:REM POSITION CURSOR
10090 PRINT A$;:REM PRINT CURRENT ENTRY
10100 POKE 204,0:REM FLASH CURSOR
10110 GET B$:IF B$="" GOTO 10110
10120 REM-RETURN KEY PRESSED?-
10130 IF B$<>CHR$(13) GOTO 10170
10140 IF LEN(A$)<L1 GOTO 10110:REM MINIMUM LENGTH TEST
10150 PRINT
10160 END:REM END OF DATA-ENTRY SEQUENCE
10170 REM-DELETE OR BACKSPACE KEY PRESSED?-
10180 IF B$<>CHR$(20) AND B$<>CHR$(157) GOTO 10220
10190 IF LEN(A$)=0 GOTO 10110:REM MINIMUM LENGTH TEST
10200 A$=LEFT$(A$,LEN(A$)-1):REM SUBTRACT RIGHT-MOST CHARACTER FROM A$
10210 GOTO 10070
10220 REM-MAXIMUM LENGTH TEST-
10230 IF LEN(A$)=L2 GOTO 10110
10240 REM-FILTER-
10250 IF ASC(B$)=44 GOTO 10300:REM COMMA ENTERED
10260 IF ASC(B$)>47 AND ASC(B$)<58 GOTO 10300:REM NUMBER
10270 IF ASC(B$)>64 AND ASC(B$)<91 GOTO 10300:REM LETTER
10280 IF ASC(B$)=32 GOTO 10300:REM SPACEBAR
10290 GOTO 10110
10300 A$=A$+B$
10310 GOTO 10070
```

Fig. 5-3."Deluxe GET" data-entry routine corresponding to logic shown in Fig. 5-2.

or GET—is best? It depends upon the program. Here is how I see it.

INPUT is fine for use in programs that have limited data-entry requirements, but not good for serious programs. The main reason is that pressing the Return key with no typed entry can disrupt the display. In addition, the "?" INPUT displays during prompting is unnecessary, and often inappropriate to the data-entry context. A prompt is not always a question. Often, it is a request for data.

INPUT# is much better than INPUT. There is less danger of display disruption and no displayed question mark. It is the preferred data-entry method for most serious programs.

GET may be used to collect a single keystroke or several keystrokes. In single-keystroke form, it is good for presenting verification statements to the user. For example, present a prompt such as the following to the user after several data entries have been made:

DO YOU WANT TO CHANGE ANYTHING? (Y/N)

The user types in a Y or N to indicate the choice.

Pressing the Return key following the entry should not be required. Single-keystroke entry such as this is not good for normal data entry—the user needs to see the data entry displayed, and be able to verify the entry with the Return key before it becomes final.

When GET is used in the multiple-keystroke, "Deluxe GET" form just described, it becomes an alternative to the INPUT or INPUT# statements. It can monitor the keystrokes one by one, filter some out, and prevent the entry from getting too long. In short, it provides complete control of the data-entry process, and provides insurance against disruption of the display. Where these factors are critical, use GET instead of INPUT or INPUT#.

Use Strings as Assignment Variables

Always use string variables in your INPUT, INPUT#, and GET statements. String variables can be anything, and your computer will not choke if the user just presses the Return key, or types a letter, number, or symbol. If you ask your computer to INPUT or GET a real variable or an integer, it is more picky. For example, if you execute this statement in your program:

```
10 INPUT A
```

and the user types in "Rose is no rose" and presses the Return key when the prompt appears, the C-64 displays the error message "REDO FROM START" and then re-displays the ?.

If the program attempts to GET a real variable, and the user types in a letter, the C-64 displays the message "SYNTAX ERROR" and crashes. Attempting to INPUT an integer variable is like playing roulette: Some numbers win, some numbers lose. Valid integer variables are between −32768 and 32767. Type in something outside this range and your C-64 goes "Aargh!" You lose.

Assign all keyboard entries to strings. If what you are looking for is a number, convert the string to its equivalent numeric value by using the VAL statement. That is, insert code such as the following:

```
10 INPUT A$
20 A=VAL(A$)
```

THE INPUT PROCESS

A well-designed data-entry routine must do more than collect the user's keystrokes and convert them to variables in memory. This is all that some data-entry routines do, and it is not enough. A good routine:

- Prompts the user—tells what entry is required.
- Collects the entries from the keyboard with a BASIC statement.
- Tests the entries.
- Permits the user to verify (observe and modify) the entries.

A poor data-entry routine omits some steps, or performs them poorly. Only when all of these pieces are in place do you have an adequate data-entry routine.

Each of these four functions—prompting, collecting keystrokes, error testing, and verification—poses certain design requirements. There are right ways and wrong ways to do each one. The discussion that follows covers each function in a separate section.

The examples use the INPUT or INPUT# statements—mainly to keep the code simple. As just discussed, it is sometimes better to use the Deluxe GET method.

Prompting

The prompt tells the user what entry to make. A prompt should do the following:

- Tell what type of entry is required. It should be both descriptive and brief. Being descriptive is more important than being brief. This prompt is both:

```
TYPE IN YOUR MONTHLY SALARY: $
```

This prompt is brief, but not descriptive enough:

```
ENTER AMOUNT:
```

Amount of what? It is not clear what is being asked for. It could be pounds, gallons, dollars, or anything measurable.

- Give the entry format, if a special one is required. For example, if a time, telephone number, or date must be entered in a special format, the prompt should show it. For example:

  TYPE IN YOUR BIRTHDATE
  (MONTH/DAY/YEAR, EXAMPLE:
  4/15/54):

  This tells the user to type in the month, day, and year, separated by slashes. Without this prompting, the date might be entered in any one of a number of different ways, such as 4,15,54; 4-15-54; 04151954; and so on.
- Attract attention—A prompt is a question and requires an answer. It cannot be answered, if the user does not see it or realize that some action is required. It must be conspicuous. A flashing cursor attracts attention and draws the eyes to the prompt. Use a flashing cursor if you can. It is also helpful to print the prompt in reverse video, or in a bright color that differs from the rest of what is on the screen. An additional way to make the prompt stand out is to print it in all capital letters.

Remember the principle of consistency. Prompt similar entries consistently. Do not ask for the same type of information in two different ways in the same program.

Collecting Keystrokes

Normally, a program collects the user's keystrokes while presenting the user with a data-entry screen. A *data-entry screen* is the display screen that the user views while typing in keyboard en-

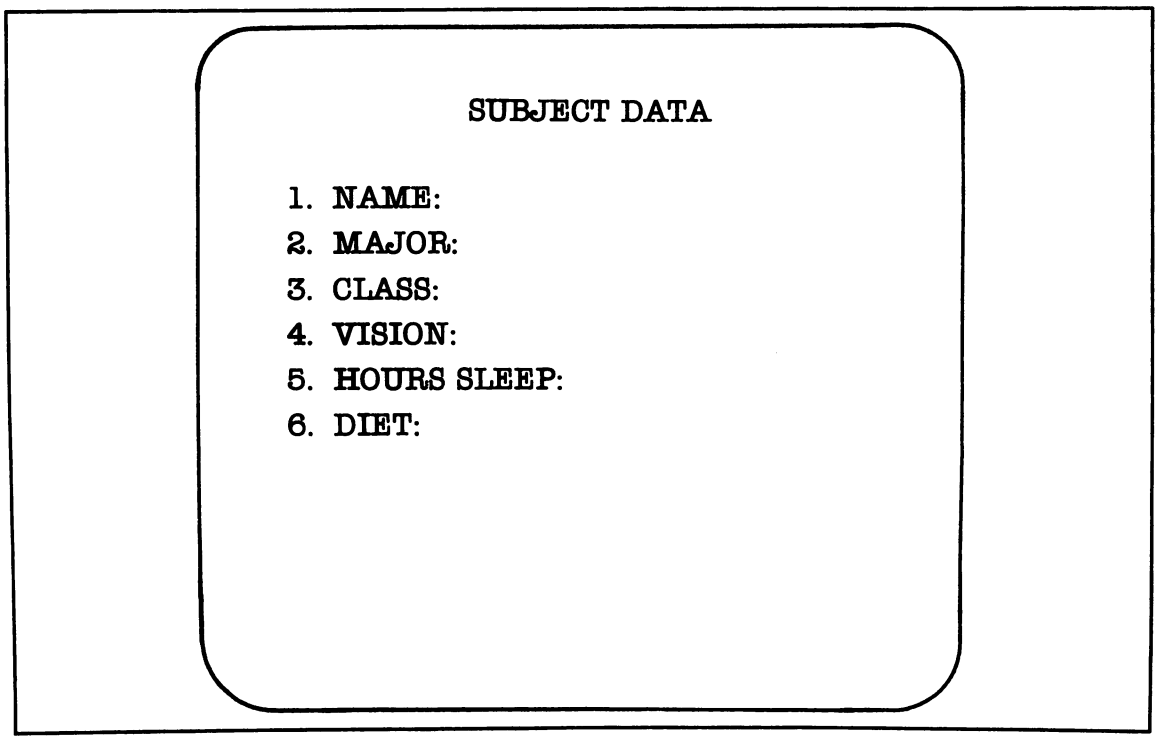


Fig. 5-4. A data-entry screen fully committed to data entry, and without displayed information.

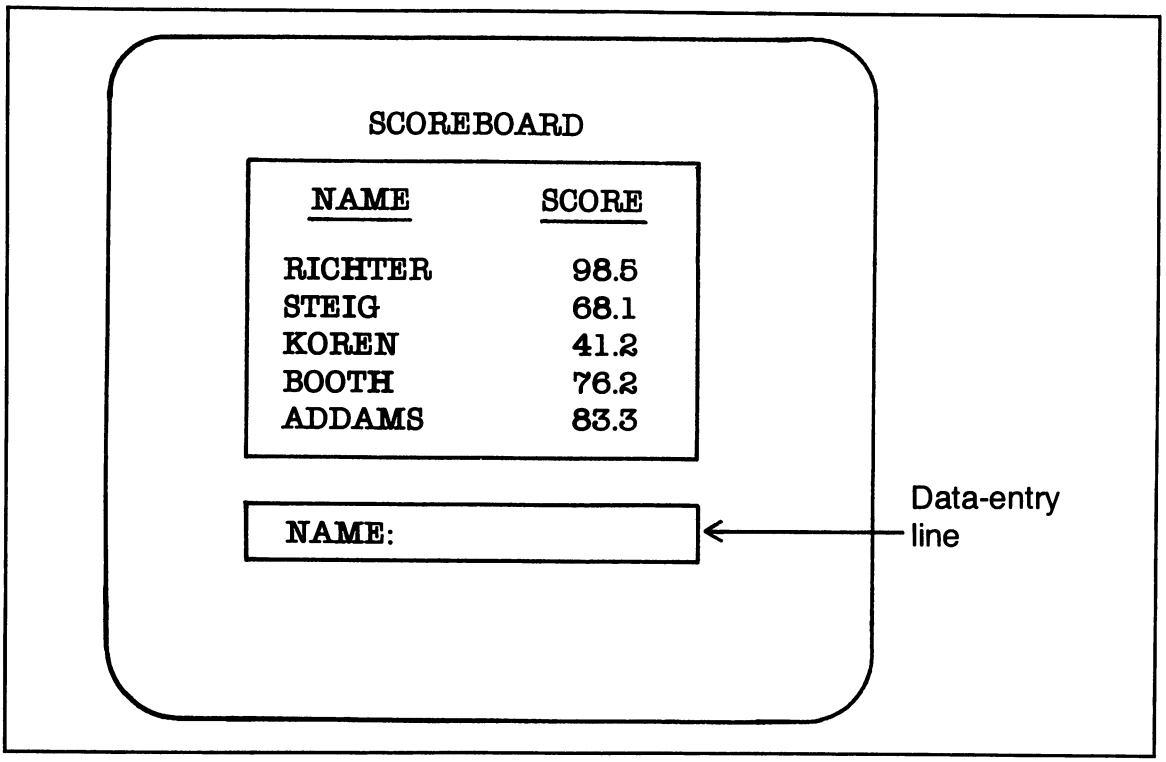


Fig. 5-5. A display screen with information at the top and a data-entry line at the bottom.

tries. Some screens are dedicated solely to this purpose (Fig. 5-4). Others may also display information that relates, directly or indirectly, to the required entries (Fig. 5-5).

Full-Screen Data Entry. You have probably used a program that has a data-entry screen similar to the one in Fig. 5-4. The screen looks like a data-entry form (Fig. 5-6). In fact, the screen may be based on a paper form, and the more the screen looks like the form, the better. Typically, the user sits before the computer and types entries in from the form. On the screen, the cursor starts at the first data-entry field. After the user types in the entry and presses the Return key, the cursor jumps to the next field. The user then types in the next entry, and so on, until the entire screen is filled in. This data-entry method is called "full-screen data entry."

Full-screen data entry is common in programs written for minicomputers or mainframes, but rarer with microcomputers. Creating such programs is usually easy with the sophisticated BASICs or other languages available in minicomputers or mainframes. It is possible, but more difficult, with a less sophisticated BASIC, such as the C-64's 2.0 level BASIC.

It is good to use full-screen data entry if users will type in data from standard forms or formatted paper copy, such as checks. Having the screen look like the copy makes it easier for the user to see the connection between the two. It is also good to use full-screen data entry if users make a series of related entries, such as a set of stock prices, and may want to refer to previous entries while making the current one. By keeping previous entries on the screen, the user can remain oriented and reduce the chances of skipping an entry or making an entry twice.

Let us go through the exercise of designing a simple data-entry screen and then creating the code necessary to generate the screen and permit full-screen data entry. The example that follows can be

extended to much more complex screens than the one described. This screen will collect three entries: NAME, AGE, and SEX, which will be assigned to the program variables NAME$, AGE, and SEX$, respectively.

To keep things simple, assume that any value of any variable is legal. This is obviously not true, but let us ignore the error tests for now; they are covered later in the chapter.

Let us start by laying out the screen on a design matrix. The result is shown in Fig. 5-7. Three prompts appear, in rows 4, 8, and 12 of column 1 of the screen.

Take it on faith for now that it is useful to create a data-entry subroutine that we can use to collect the keystrokes. Figure 5-8 is the listing of a subroutine that will do the job for us. It makes use of the SYScalls described in Chapter 4, although the cursor could also be positioned and the line cleared with BASIC. The arguments of this subroutine are V, H, and P$. V is the number of the row, and H is the number of the column at which the prompt is to

SUBJECT QUESTIONNAIRE

Please write in the answers to the following questions.

1. **Name (last, first)** _____

2. **Major** _____

3. **Class (check one)**
   - **Freshman** ____
   - **Sophomore** ____
   - **Junior** ____
   - **Senior**____

4. **Vision (check one)**
   - **Normal** ____
   - **Corrected** ____

5. **Hours sleep per night** _____

6. **Diet (check one)**
   - **None** ____
   - **High carbohydrate** ____
   - **High protein** ____

Fig. 5-6. A questionnaire corresponding to the data-entry screen shown in Fig. 5-4.

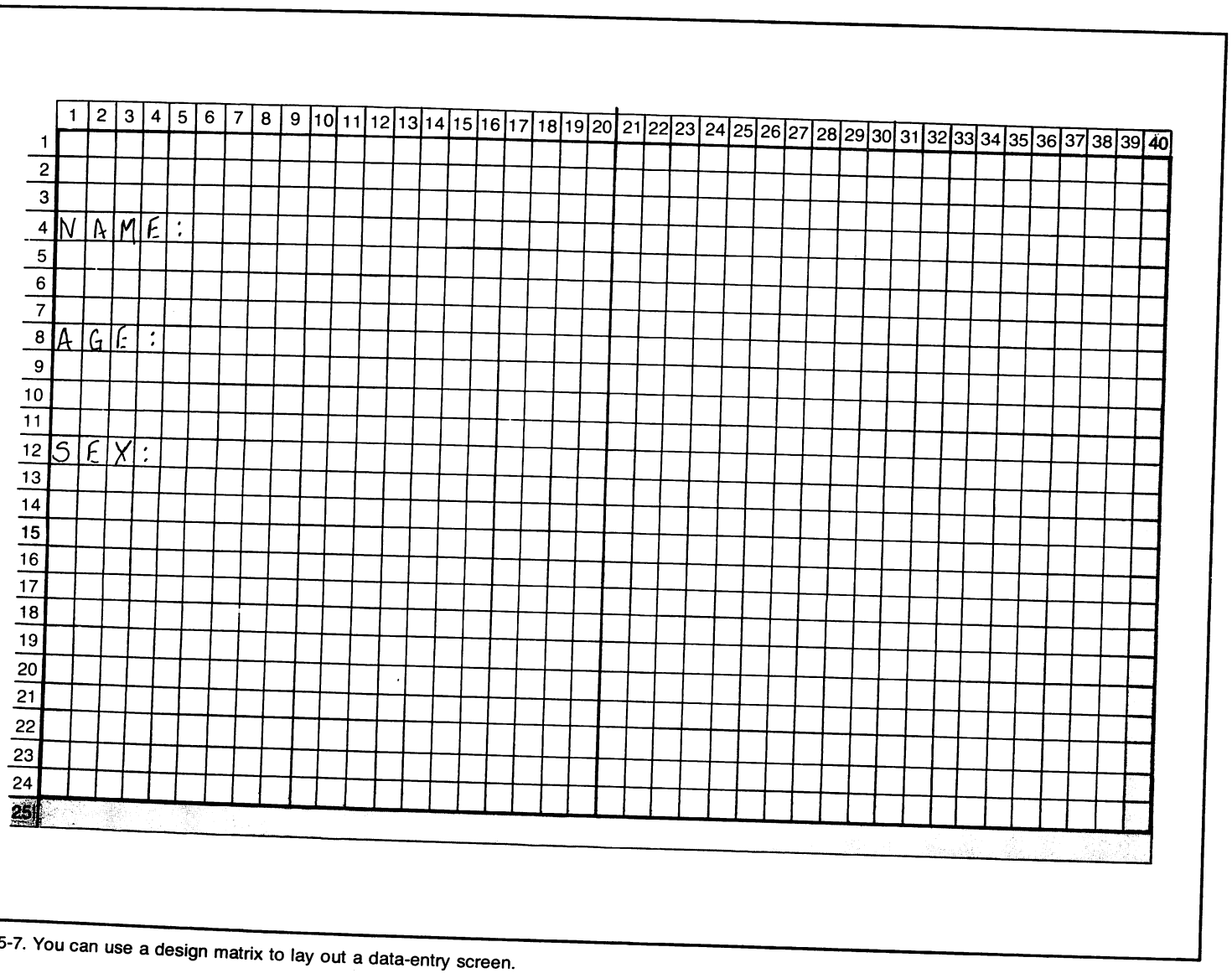


Fig. 5-7. You can use a design matrix to lay out a data-entry screen.

```
10 REM ************************************
20 REM * NOTE: THIS SUBROUTINE REQUIRES   *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
3010 REM--INPUT# DATA ENTRY--
3020 SYS C0,1,V,H :REM CLEAR LINE
3030 SYS C0,0,V,H:REM POSITION PROMPT
3040 PRINT CHR$(18);:REM REVERSE ON
3050 PRINT P$+":";:REM PRINT PROMPT
3060 PRINT CHR$(146);:REM REVERSE OFF
3070 OPEN 1,0
3080 INPUT#1,A$
3090 PRINT
3100 CLOSE 1
3110 RETURN
```

Fig. 5-8. A subroutine to collect keystrokes with the INPUT# statement.

be printed. P$ is the prompt. When these arguments are defined, and the subroutine is called, the prompt is printed in reverse video at the designated location, and the INPUT# statement collects the data entry for us, returning it to us as the variable A$.

Here is how the subroutine works. Line 3020 clears the prompt line, and 3030 positions the cursor before printing the prompt. Line 3040 turns on reverse video and line 3050 prints the prompt, attaching a colon to it. Line 3060 turns reverse video off. Lines 3070-3100 collect the keystrokes, using the INPUT# statement, and assigning them to the variable A$.

Now let us create the code to generate the data-entry screen. Since all of the prompts must appear on the screen when the data-entry routine begins, we must print each prompt twice. We will start by clearing the screen, and then print all of the prompts for the entire screen. Then we will set up and call the subroutine to display the first prompt and collect its keystrokes, do the same for the second prompt, and repeat for the third. (By redesigning the data-entry subroutine not to display the prompt, we could save having to print prompts twice, but it is handy to have it print the prompt in other types of data-entry situations.)

Figure 5-9 shows the code required to generate the data-entry screen and collect the data entries. The data-entry subroutine runs from lines 3000-3110. Line 10020 clears the screen. Line 10030 positions the cursor, and 10040 PRINTs the first prompt in reverse video. Lines 10050 and 10060 do the same for the second prompt, and 10070 and 10080 for the third. After these lines have been executed, the data-entry screen has been created.

Lines 10110, 10120, and 10130, respectively, define the row (V) and column (H) in which the prompt is to appear, and what the prompt is to say (P$). Line 10140 calls the data-entry subroutine. After it has been executed, the value it returns is assigned in line 10150 to the program variable NAME$. Lines 10160-10200 do likewise for AGE, and lines 10210-10250 for SEX$.

Single-Line Data Entry. A much simpler data-entry method is to take all entries on a single line of the screen. Not only is this method easier to code, but it makes sense if the user might need to refer to information on another part of the screen. For example, the screen might contain a directory of names, a list of parts, or something else that could influence the entries.

Writing a program to do single-line data entry is fairly easy, especially if we use a data-entry subroutine such as the one in Fig. 5-8 (it is no longer

necessary to accept the value of this subroutine on faith). Figure 5-10 is the listing of a program that collects the same three variables—NAME, AGE, SEX—as the full-screen data-entry program described earlier. The current program will, however, display the prompts, and collect the entries, one at a time, from row 22 of the screen.

The main program consists of lines 10000-10150. Line 10010 clears the screen. Lines 10030-10050 define the arguments V, H, and P$ for the first variable. Line 10060 calls the data-entry subroutine, and line 10070 assigns the value of A$ returned by the subroutine to the program variable NAME$. Lines 10080-10110 do the same thing for

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
3010 REM--INPUT# DATA ENTRY--
3020 SYS C0,1,V,H :REM CLEAR LINE
3030 SYS C0,0,V,H:REM POSITION PROMPT
3040 PRINT CHR$(18);:REM REVERSE ON
3050 PRINT P$+":";:REM PRINT PROMPT
3060 PRINT CHR$(146);:REM REVERSE OFF
3070 OPEN 1,0
3080 INPUT#1,A$
3090 PRINT
3100 CLOSE 1
3110 RETURN
10000 REM--FULL SCREEN DATA ENTRY PROGRAM--
10010 REM-INITIALIZE SCREEN-
10020 PRINT CHR$(147)
10030 SYS C0,0,4,1
10040 PRINT CHR$(18)"NAME:"
10050 SYS C0,0,8,1
10060 PRINT CHR$(18)"AGE:"
10070 SYS C0,0,12,1
10080 PRINT CHR$(18)"SEX:"
10090 REM-PROMPT AND COLLECT KEYSTROKES-
10100 REM-NAME-
10110 V=4
10120 H=1
10130 P$="NAME"
10140 GOSUB 3010
10150 NAME$=A$
10160 REM-AGE-
10170 V=8
10180 P$="AGE"
10190 GOSUB 3010
10200 AGE=VAL(A$)
10210 REM-SEX-
10220 V=12
10230 P$="SEX"
10240 GOSUB 3010
10250 SEX$=A$
10260 END
```

Fig. 5-9. Full-screen data-entry program based on screen design shown in Fig. 5-7.

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
3010 REM--INPUT# DATA ENTRY--
3020 SYS C0,1,V,H :REM CLEAR LINE
3030 SYS C0,0,V,H:REM POSITION PROMPT
3040 PRINT CHR$(18);:REM REVERSE ON
3050 PRINT P$+":";:REM PRINT PROMPT
3060 PRINT CHR$(146);:REM REVERSE OFF
3070 OPEN 1,0
3080 INPUT#1,A$
3090 PRINT
3100 CLOSE 1
3110 RETURN
10000 REM--ONE-LINE DATA-ENTRY PROGRAM--
10010 PRINT CHR$(147):REM CLEAR SCREEN
10020 REM-NAME-
10030 V=22
10040 H=1
10050 P$="NAME"
10060 GOSUB 3010
10070 NAME$=A$
10080 REM-AGE-
10090 P$="AGE"
10100 GOSUB 3010
10110 AGE=VAL(A$)
10120 REM-SEX-
10130 P$="SEX"
10140 GOSUB 3010
10150 SEX$=A$
10160 END
```

Fig. 5-10. Single-line data-entry program.

the AGE variable, and lines 10120-10150 for the SEX variable.

The Scrolling-Screen Method. Now we come to what is probably the most common method of taking data entries. Try to remember back to the first program you ever wrote for a taking a series of data entries. It may very well have consisted of a series of INPUT statements such as these:

```
10 INPUT "NAME";N$
20 INPUT "AGE";AGE$
30 INPUT "SEX";SEX$
```

What happens when the program executes depends mainly on where the first prompt appears. If it starts at the top of the screen (Fig. 5-11), then subsequent prompts work their way down the screen toward the bottom, one below the other. If it starts at the bottom (Fig. 5-12), then old prompts scroll up each time a new prompt appears. While this data-entry method is simple, it produces either a shifting center of focus, as the prompt line moves down the screen, or undesirable scrolling of the screen. More bluntly, it makes for sloppy data-entry programs. Avoid it if you can.

Error-Testing

Most data-entry routines require error tests.

An *error test* is code that examines the user's data entry according to certain rules to determine whether that entry makes sense. The simple data-entry programs we have been working with have three data entries: NAME, AGE, and SEX. Although we did not build error tests into the programs described earlier, we would certainly do so in any program that we wanted to use to collect valid data. The types of error tests depend upon the nature of the data. For example;

- We would conduct a length test on NAME to assure that the number of characters in the name fell within certain limits, such as 2 and 24 characters.
- We would conduct a range test on AGE to assure that the value fell between, say, 2 and 99 years.
- Sex would have to be either M or F.

If we did not conduct such tests, data-entry errors made by users would find their way into our data base, and cause problems later. The best time to catch such errors is when the user types in the data. So we must write code that detects the errors.

That alone is not enough, however. When an error occurs, we must tell the user about it, and provide enough additional information so that the

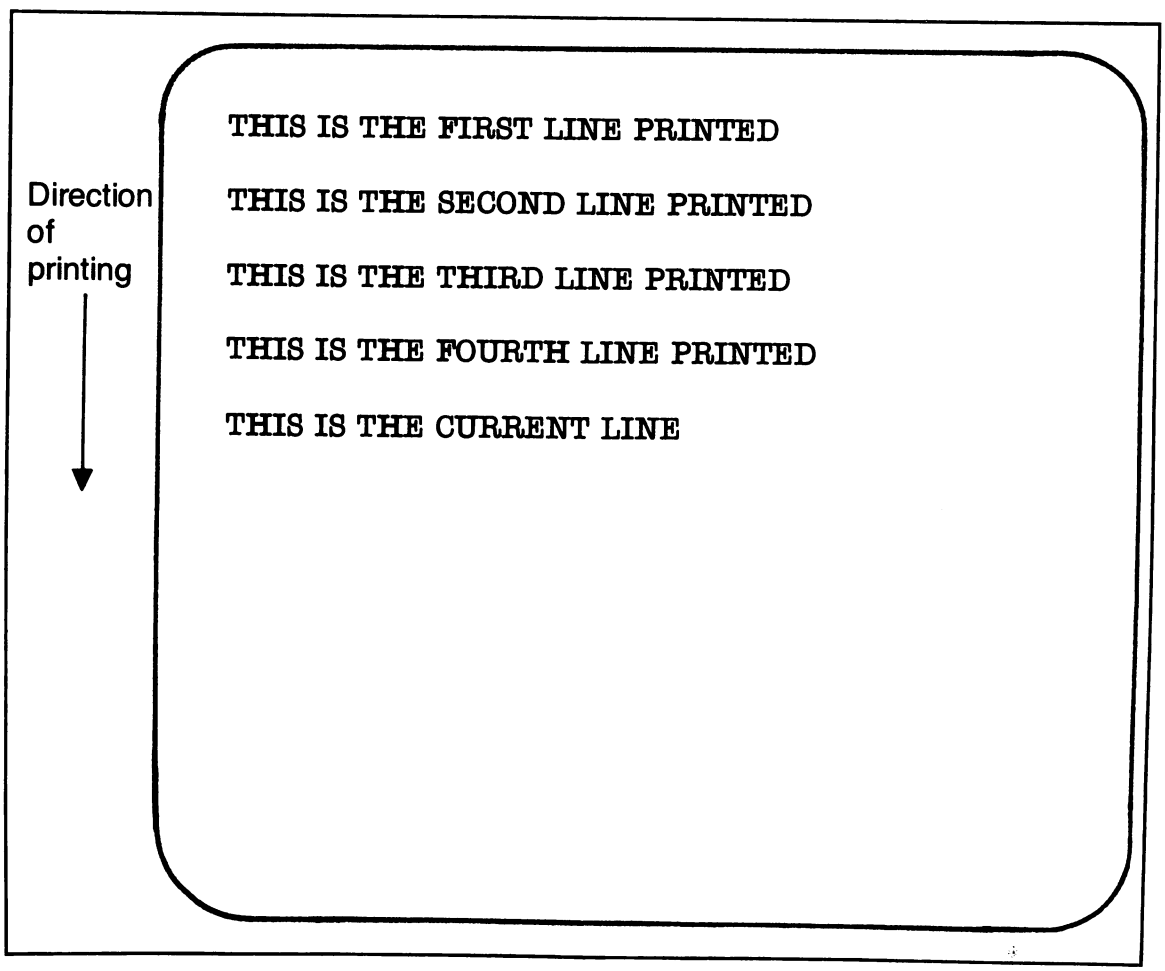


Fig. 5-11. If the screen is cleared and then data entries are taken without controlling the location of the prompts, each successive prompt will appear one row further down the screen.

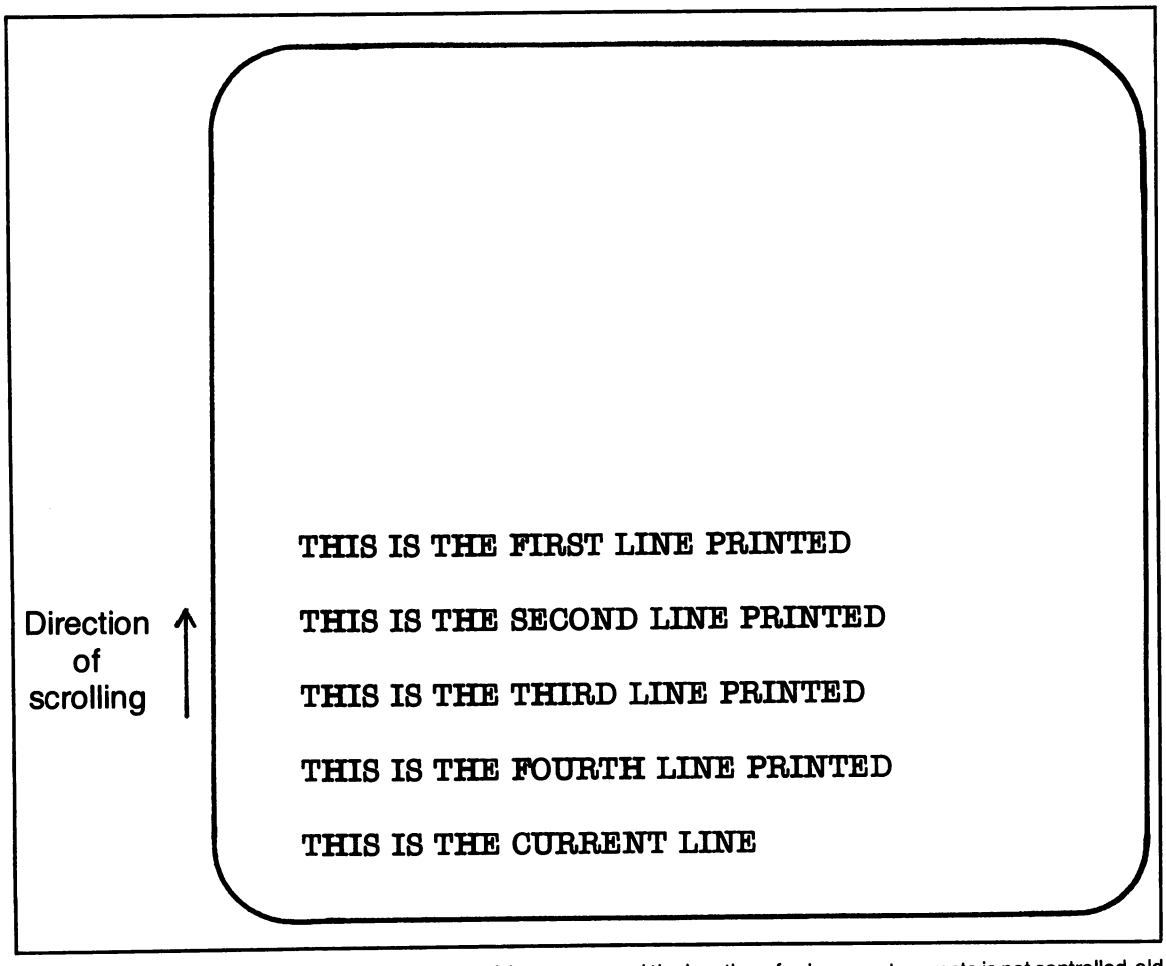


Fig. 5-12. If the first prompt is printed at the bottom of the screen, and the location of subsequent prompts is not controlled, old prompts scroll up the screen as each new prompt is printed on the bottom row.

error can be corrected. This is what error messages are for. In this section, then, we are concerned with two issues: (1) error detection, and (2) error messages. These two factors together comprise error-testing.

Error Detection. The simplest type of error test is the identity test. For example, in this prompt:

PLEASE TYPE IN YOUR SEX ('M' OR 'F'):

the only acceptable entries are M or F. The code to generate the prompt and conduct the test is as follows:

```
10 INPUT"PLEASE TYPE IN YOUR SEX ('M' OR 'F'): ":SEX$
20 IF SEX$ < > "M" AND SEX$ < > "F" GO-TO 10
```

The identity test is useful in some data-entry programs. More often than not, however, we want to know whether a value falls within a particular range.

Suppose that you write a simple data entry program that looks lke this:

```
10 INPUT"AGE:";A$
20 AGE=VAL(A$)
```

This particular prompt is looking for someone's age. An excited user might press the Return key without making a numeric entry. The value of AGE returned by the VAL function would then be zero. The value zero would be returned if the user typed in a letter or symbol key. Alternatively, if the user struck the key too many times, a very large number might result. The user's age of 22 might, for example, be typed in as 222. It is fairly easy to check the numeric range of entries and reject those that are outside of it. We must, of course, set a lower and upper limit on what will be acceptable. Assume that this limit is from 2 to 99 years. By adding the following line to our program, we can have it make a range test:

```
30  IF AGE < 2 OR AGE > 99 GOTO 10
```

Line 30 tests whether AGE is less than 2 or greater than 99. If so, then it is outside the legal range and control is sent back to line 10, which forces the user to reenter the value. If the value is within the acceptable range, then the routine is finished.

The lines of the code that perform error tests invariably make use of logical and relational operators: =, <, >, AND, OR, and NOT. Simple tests (such as the range test just illustrated) involve simple expressions. Sometimes, however, the expressions used in a test become quite complicated. For example, if the acceptable range of entries is disjointed—ages from 1-23, from 43-53, or age 99 are acceptable—then the logical expression for making the test gets a little hairy:

```
30  IF NOT ((AGE> = 1 AND AGE < = 23)OR
    (AGE > = 43 AND AGE < = 63) OR
    AGE=99) GOTO 10
```

The lesson is that, if you are not up on your operators, and you do no need to perform complex error tests, do a little homework. In what follows, I assume that you have a good understanding of the logic involved, and will let the logical expressions stand alone, without explanation.

We just examined the simplest type of range test, one that tests whether the data entry falls between a lower and upper numeric bound. Range tests are not limited to numbers. Every character that can be displayed on your computer has an *ASCII* (American Standard Code for Information Interchange) code. This is a number that amounts to the character's pecking order in the character set. The ASCII values for your C-64's character set are given in Appendix F of your Commodore 64 User's Guide. Use this appendix to find the ASCII values of the characters $, &, 4, and E.

The values are 37, 38, 52, and 69. The character with the highest ASCII value is E and that with the lowest is %. Your computer knows the ASCII value of each character. You can use the ASC statement to obtain the ASCII value of any character. For example, to find the ASCII value of the character %, type in the following statement:

```
PRINT ASC("%")
```

Now type this in:

```
PRINT"E" > "%"
```

The result you get from this is −1, which means that character E is greater than character %. Your computer arrived at this answer by comparing the ASCII values of the two characters. Since the ASCII value of E is greater than the ASCII value of %, the statement is true, and its logical result is −1.

What all this is leading up to is that you can perform a range test not just on numbers, but also on characters and symbols. To illustrate, suppose, for the sake of argument, that you want to design a data-entry routine that accepts any letter between A (ASCII value 65) and E (ASCII value 69). The code to perform this test is as follows:

```
10  INPUT "LETTER:";A$
20  IF A$ <"A" OR A$ > "E" GOTO 10
```

To make it more interesting, suppose that acceptable entries are characters A through E or numbers 1 through 4. The code to perform this test is:

```
10  INPUT"CHARACTER:";A$
20  IF NOT ((A$>="A" AND A$<="E")
OR A$>="1" AND A$<="4")) GOTO
10
```

(This test illustrates what I mean about the importance of feeling comfortable with operators.

Note that tests such as these can be posed in either of two ways:

- Detect an unacceptable condition.
- Detect an acceptable condition.

For example, in this program:

```
10 INPUT LETTER:";A$
20 IF A$<"A" OR A$>"E" GOTO 10
```

the test looks for an unacceptable condition—a character outside the range. We could change the test to look for an acceptable condition by rewriting line 20 to read:

```
20  IF A$>"@" AND A$<"F" THEN (continue
program)
```

If you want to bounce control to a particular place when an error occurs, then it is best to look for the unacceptable condition. If you detect an acceptable condition, as in the most recent line 20, then you need a separate line to handle the error condition. The following listing illustrates an alternate version of the earlier program. The error test in this one looks for the acceptable condition:

```
10  INPUT "LETTER:";A$
20  IF A$>="A" AND A$<="E" GOTO 40
30  GOTO 10
40  END
```

As you are aware, you can change a positive test to a negative test by adding NOT at the beginning. For example, IF NOT (A$ > " " AND A$ < "F") is the same as IF A$<"A" OR A$>"E".

You can use character range tests on entries that are longer than a single character. When BASIC evaluates a character string, it looks at only the first character in the string. To illustrate, suppose that we want to design a data-entry routine for collecting the names of people, but only if their names fall between the letters A and E in the alphabet. The little program we have been using can be used for this purpose. Type in the last program, RUN it, and then type in some names that start with letters between A and E, and some that do not.

In short, it is not necessary to worry about how many characters are in the entry that is keyed in. The test looks only at the first character. There are methods to look inside the entry, but to do so you must take it apart using string-manipulation routines.

Another common test is the length test. It is used to ensure that the number of characters in a typed entry does not fall below or exceed a certain limit, such as between 2 and 24 characters. (Incidentally, any time you have a user type in a string that you intend to store on disk, make sure that it is at least one character long. Disk does not recognize the null string as a legitimate character and does not store it. See Chapter 8 for details.).

The simplest way to perform a length test is by using the LEN statement in code such as the following:

```
10  INPUT "NAME (2-24 CHAR):";A$
20  IF LEN(A$) < 2 OR LEN(A$) > 24 GO-
TO 10
```

The program permits the user to type in any string. Line 20 uses the LEN function to calculate the length of the string. If the length is shorter than 2 characters or greater than 20 characters, then control returns to line 10. As you can see, the length test works much like the range test.

These examples illustrate the basic concepts underlying error tests. Of course, the examples are simplified and do not tell the whole story. The code used the INPUT statement (you know what its shortcomings are), and if an error condition occurred, control bounced back to re-present the prompt, causing scrolling. We can make everything much neater by using the INPUT# routine de-

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
3010 REM--INPUT# DATA ENTRY--
3020 SYS C0,1,V,H :REM CLEAR LINE
3030 SYS C0,0,V,H:REM POSITION PROMPT
3040 PRINT CHR$(18);:REM REVERSE ON
3050 PRINT P$+":";:REM PRINT PROMPT
3060 PRINT CHR$(146);:REM REVERSE OFF
3070 OPEN 1,0
3080 INPUT#1,A$
3090 PRINT
3100 CLOSE 1
3110 RETURN
10000 REM--FULL SCREEN DATA ENTRY PROGRAM--
10010 REM-INITIALIZE SCREEN-
10020 PRINT CHR$(147)
10030 SYS C0,0,4,1
10040 PRINT CHR$(18)"NAME:"
10050 SYS C0,0,8,1
10060 PRINT CHR$(18)"AGE:"
10070 SYS C0,0,12,1
10080 PRINT CHR$(18)"SEX:"
10090 REM-PROMPT AND COLLECT KEYSTROKES-
10100 REM-NAME-
10110 V=4
10120 H=1
10130 P$="NAME"
10140 GOSUB 3010
10150 NAME$=A$
10151 REM-ERROR TEST-
10152 IF LEN(NAME$)>=2 AND LEN(NAME$)<=24 GOTO 10160
10154 GOTO 10140
10160 REM-AGE-
10170 V=8
10180 P$="AGE"
10190 GOSUB 3010
10200 AGE=VAL(A$)
10201 REM-ERROR TEST-
10202 IF AGE>=2 AND AGE <=99 GOTO 10210
10205 GOTO 10190
10210 REM-SEX-
10220 V=12
10230 P$="SEX"
10240 GOSUB 3010
10250 SEX$=A$
10251 REM-ERROR TEST-
10252 IF SEX$="M" OR SEX$="F" GOTO 10260
10255 GOTO 10240
10260 END
```

Fig. 5-13. Full-screen data-entry program based on Fig. 5-9, but with simple error testing.

scribed earlier in this chapter, since it positions the prompt before it PRINTS it, thereby preventing scrolling.

Figure 5-13 is a version of the full-screen data-entry program presented earlier in Fig. 5-9. A few lines have been added to the program to provide error-testing.

Line 10152 conducts a length test on NAME$; line 10202 conducts a range test on AGE, and line 10252 conducts an identity test on SEX$. These tests are straightforward enough. If the test fails, however, control now jumps back to a line that regenerates the prompt at a specific location on the screen. When you handle error-testing this way, there is no need for the screen to scroll.

Error Messages. Detecting errors is fine, but it is not quite enough to stay on good terms with program users. They appreciate it when you keep them from filling their data bases with garbage, but they do not appreciate it when they respond to a prompt, type in an invalid entry, and the prompt keeps reappearing. Almost everyone who has used a computer has been in a situation where they faced a prompt line, typed something in, and the computer did not accept it and kept repeating the prompt. In these situations, the computer sometimes displays a cryptic message such as "ILLEGAL INPUT" or "SYSERR 401" that makes some users want to perform an act of violence upon the programmer.

What is lacking in these situations is a good error message. Each time the user types in something that is invalid, display an error message before repeating the prompt. The message should do these things:

- Attract attention—The best way is to make it flash.
- Define the error—Tell the user what is wrong with the entry.
- Give the recovery action, if it is something other than retyping the entry.

Display the error message close to the prompt so that the user can relate it to the data entry. Let us concentrate for a moment on the mechanical aspects of displaying error messages by devising a subroutine to display a flashing error message at a definable location on the screen. After doing this, I will discuss in more detail what an error message should contain.

Figure 5-14 contains the listing of a subroutine for displaying an error message. The subroutine consists of lines 3200-3310, and this subroutine also requires that the Time Delay subroutine (lines 4100-4130) be present. The arguments of the subroutine are V, H, and ER$, which are the row number, column number, and error message, respectively. The subroutine clears whatever line it is on from the column number to the right edge of the display—the error message can be as long as the space available. If the entire row is cleared, the message can be up to 39 characters long (40 characters would cause an unwanted line feed).

Here is how the subroutine works.

Line 3210 clears the display row.

Line 3220 begins a FOR-NEXT loop that terminates on line 3290. Within this loop, the error message is alternately positioned and then printed for .2 second first in reverse video and then .2 seconds in normal video. The cycle repeats 10 times, displaying the flashing message for a total of about four seconds. Line 3230 sets a display time of .2 seconds and then calls the delay subroutine at line 4100. Line 3240 positions the cursor. Line 3250 PRINTs ER$ in reverse video. Line 3260 then introduces another .2 second delay; line 3270 positions the cursor again, and line 3280 PRINTs the message in normal video.

This subroutine can be used to PRINT a flashing message anywhere on the screen, and it is not restricted to error messages. It is also useful for attracting attention to other matters of concern. Note, however, that while this subroutine is being executed, everything else going on in the program stops. Also, when it finishes executing, whatever message it printed will remain on the screen in normal video until it is erased or overprinted.

To illustrate the subroutine's use, insert these lines into the listing shown in Fig. 5-14 and try it out:

```
990  GOTO 10000
```

```
10000  INPUT "V:";V
10010  INPUT "H:";H
10020  INPUT "ER$:";ER$
10030  GOSUB 3200
10040  GOTO 10000
```

Now that you see how the subroutine works, let us add it to the data-entry program in Fig. 5-13 so that we can generate error messages in response to invalid data entries. Figure 5-15 is the data-entry program we examined earlier, but with a few changes.

- Two subroutines have been added: Error Message (lines 3200-3310) and Time Delay (lines 4100-4130).
- Error messages and subroutine calls have been added at lines 10153, 10154; 10203, 10204; and 10253, 10254.

This program permits full-screen data entry. If an invalid entry is typed in, an error message is printed in place of the prompt for about four seconds, and then the prompt reappears, permitting reentry of the data.

Let us see how the code works by examining the data-entry steps for the first variable, NAME$. Most of this code was described earlier; so let us focus on how the error message is generated. Line 10152 performs the error test. If this test is passed, then control is sent to line 10160, permitting entry of the next variable. If the test is failed, then control drops first to line 10153, which defines the error message ER$, and then to line 10154, which calls the Error-Message subroutine. This subroutine displays the message at row V and column H (these two arguments were defined in lines 10110 and 10120 and are the same for both the INPUT# and Error-Message subroutines). After the RETURN from the subroutine, control drops to line 10155, which sends it back to line 10140, where the INPUT# subroutine again presents the data-entry prompt.

There are a lot of steps in this process, but nothing earth-shakingly complex. It it is fairly easy to build such error-messaging into your programs

```
10 REM ************************************
20 REM * NOTE: THIS SUBROUTINE REQUIRES  *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE    *
40 REM * LOADER (FIGURE 4-6)             *
50 REM ************************************
3200 REM--ERROR MESSAGE--
3210 SYS C0,1,V,H:REM CLEAR LINE
3220 FOR B=1 TO 10
3230 TX=.2:GOSUB 4100:REM .2 SEC DELAY
3240 SYS C0,0,V,H:REM POSITION CURSOR
3250 PRINT CHR$(18)ER$:REM PRINT ERROR MESSAGE IN REVERSE VIDEO
3260 TX=.2:GOSUB 4100:REM .2 SEC DELAY
3270 SYS C0,0,V,H
3280 PRINT CHR$(146)ER$:REM PRINT ERROR MESSAGE IN NORMAL VIDEO
3290 NEXT
3300 SYS C0,1,V,H
3310 RETURN
4100 REM--TX SECOND DELAY--
4110 FOR A=1 TO TX*870
4120 NEXT
4130 RETURN
```

Fig. 5-14. Subroutine (lines 3200-3310) for generating a flashing error message at a designated row and column.

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
3010 REM--INPUT# DATA ENTRY--
3020 SYS C0,1,V,H :REM CLEAR LINE
3030 SYS C0,0,V,H:REM POSITION PROMPT
3040 PRINT CHR$(18);:REM REVERSE ON
3050 PRINT P$+":";:REM PRINT PROMPT
3060 PRINT CHR$(146);:REM REVERSE OFF
3070 OPEN 1,0
3080 INPUT#1,A$
3090 PRINT
3100 CLOSE 1
3110 RETURN
3200 REM--ERROR MESSAGE--
3210 SYS C0,1,V,H:REM CLEAR LINE
3220 FOR B=1 TO 10
3230 TX=.2:GOSUB 4100:REM .2 SEC DELAY
3240 SYS C0,0,V,H:REM POSITION CURSOR
3250 PRINT CHR$(18)ER$:REM PRINT ERROR MESSAGE IN REVERSE VIDEO
3260 TX=.2:GOSUB 4100:REM .2 SEC DELAY
3270 SYS C0,0,V,H
3280 PRINT CHR$(146)ER$:REM PRINT ERROR MESSAGE IN NORMAL VIDEO
3290 NEXT
3300 SYS C0,1,V,H
3310 RETURN
4100 REM--TX SECOND DELAY--
4110 FOR A=1 TO TX*870
4120 NEXT
4130 RETURN
10000 REM--FULL SCREEN DATA ENTRY PROGRAM--
10010 REM-INITIALIZE SCREEN-
10020 PRINT CHR$(147)
10030 SYS C0,0,4,1
10040 PRINT CHR$(18)"NAME:"
10050 SYS C0,0,8,1
10060 PRINT CHR$(18)"AGE:"
10070 SYS C0,0,12,1
10080 PRINT CHR$(18)"SEX:"
10090 REM-PROMPT AND COLLECT KEYSTROKES-
10100 REM-NAME-
10110 V=4
10120 H=1
10130 P$="NAME"
10140 GOSUB 3010
10150 NAME$=A$
10151 REM-ERROR TEST-
10152 IF LEN(NAME$)>=2 AND LEN(NAME$)<=24 GOTO 10160
10153 ER$="NAME MUST BE 2-24 CHARACTERS LONG"
10154 GOSUB 3200
10155 GOTO 10140
10160 REM-AGE-
10170 V=8
10180 P$="AGE"
10190 GOSUB 3010
```

```
10200 AGE=VAL(A$)
10201 REM-ERROR TEST-
10202 IF AGE>=2 AND AGE <=99 GOTO 10210
10203 ER$="AGE MUST BE BETWEEN 2 AND 99 YEARS"
10204 GOSUB 3200
10205 GOTO 10190
10210 REM-SEX-
10220 V=12
10230 P$="SEX"
10240 GOSUB 3010
10250 SEX$=A$
10251 REM-ERROR TEST-
10252 IF SEX$="M" OR SEX$="F" GOTO 10260
10253 ER$="PLEASE ENTER 'M' OR 'F'"
10254 GOSUB 3200
10255 GOTO 10240
10260 END
```

Fig. 5-15. Full-screen data-entry program based on Fig. 5-13, but with the addition of error messages.

by using these procedures. Even if you can present such messages, however, you must know what to say to avoid that "INVALID ENTRY" and other such insensitive and antisocial stuff.

How do you write a good error message? A simple question, right? Well, the answer may be simple, but it is not obvious. $E = mc^2$ is simple, too, but it took Albert a lot of work to get there. Admittedly, writing error messages is not in the same league as relativity, but you get the idea.

Make your error messages short, but be sure they tell the story. Do not leave the user guessing. Avoid sarcastic messages. They will turn program users against both you and the program. It is also not generally a good idea to use humor in your error messages, either. The message you find in your first fortune cookie is fun, but if you keep finding it, it kind of gets on your nerves.

The error message must alert the user that something is wrong, identify the problem, and, if it is not obvious, tell what to do about it. Flashing usually takes care of the alerting part, but if you are not sure this will do the job, combine it with sound. Have your computer beep or make some other kind of pained noise when something goes wrong.

You must take care of identifying the error and telling what to do about it with the written message. The first part of this—identifying the error—is required. All of the messages used in our example identify the errror condition. Here they are:

- Name test: NAME MUST BE 2-24 CHARACTERS LONG
- Age test: AGE MUST BE BETWEEN 2 and 99 YEARS
- Sex test: PLEASE ENTER EITHER "M" OR "F"

The first two messages identify the condition but do not tell what to do, since it can be considered obvious. The third message tells what to do and implicitly defines the error. Telling what to do may be considered optional in most data-entry programs since the ongoing activity is date entry and a data-entry error calls for reentry.

In some programs, telling the corrective action is more important. This is often the case when the user is engaged in keying in entries that control the program and may have gotten the program into a state that requires some non-obvious action for recovery.

Error messages are a form of written communication, and the types of messages you provide tell a lot about your personality. We all have recollections of messages (or nonmessages) we have encountered in our work with computer programs.

The way error-messaging is handled provides insight into the programmer's mentality, and is somewhat like handwriting analysis. If your program provides informative error messages, it is rather like the flowing and elliptical handwriting of John Hancock. If the error messages are cryptic, uninformative, or insulting, they are like the handwriting of someone with a personality disorder who writes in little broken, disconnected letters.

Verification

To *verify* something is to confirm its truth. In a data-entry program, verification occurs when the user views an entry that has been typed in and gives it the okay. Until that point, the entry is not finalized. When the user verifies it, it stops being a potential entry and becomes an actual entry. Verification is important because program users often make mistakes or simply change their minds. They need to be able to change their entries.

The three most common verification techniques used in microcomputer programs are the following:

- Return-key verification.
- Line verification.
- Page verification.

Return-Key Verification. The simplest verification technique is Return-key verification. You use this all the time and probably do not think of it as a verification technique. It is built into the INPUT and INPUT# statements. Whenever your program executes one of these statements, it requires the user to type in an entry and follow it by pressing the Return key. The entry is not final until the Return key is pressed. The Return key, in other words, verifies what has been typed.

Return-key verification allows the user to recognize and modify entries before the Return key has been pressed, but not afterward. Still, it is desirable, and you should use it for entries that the user may want to change—such as data entries or options on a program control menu.

Return-key verification does not require the use of the INPUT or INPUT# statements. It can also be built into data-entry routines that use the GET statement, such as the "Deluxe GET" method described earlier in this chapter. The GET statement by itself does not have Return-key verification—whatever key is pressed is assigned immediately to the relevant variable, without giving the user a chance to verify it. For this reason, the naked GET statement is not good as a way of taking data entries.

Line Verification. Line verification consists of displaying a verification prompt after an entry that permits the user to back up and change the entry. A prompt such as this appears:

```
DO YOU WANT TO CHANGE YOUR ENTRY? (Y/N)
```

If the user types in Y, then control is sent back to the data-entry prompt and permits the user to reenter the item. If the user types in N, then the entry is verified and the program proceeds.

To illustrate, the following program collects a single data entry and then permits the user to verify it:

```
10  INPUT"NAME: ";NAME$
20  PRINT"OKAY? (Y/N) ";
30  GET A$:IF A$="" GOTO 30
40  PRINT
50  IF A$="N" GOTO 10
60  IF A$="Y" THEN END
70  GOTO 20
```

Line 10 collects the data entry. Line 20 PRINTS the prompt "OKAY? (Y/N)". Line 30 then GETs a single character. Line 50 tests whether the entered character is an N and, if so, sends control back to line 10, which permits the user to reenter the data. If a Y is typed in, the program ends (line 60). If neither Y nor N is typed in, line 70 sends control back to line 20, representing the verification prompt.

The nice thing about line verification is that it allows the user to make a change after the Return key has been pressed.

Line verification works well if there are only a

few entries to verify, but imagine what it would be like to make 20 entries and verify each one. It all becomes very tedious very quickly, and the program moves along very slowly. If a program requires many data entries that must be verified, it is better to use page verification.

Page Verification. Page verification is similar to line verification except that the user makes a series of entries before verifying them. It is often used with full-screen data entry. After the last data-entry field is completed, a prompt such as the following appears on the display:

DO YOU HAVE CHANGES? (Y/N)

If the entries are okay, the user types in N and the program proceeds. If not, the user types in Y and a prompt such as the following appears:

SELECT ITEM # TO CHANGE

The user then types in the number of the item, and the cursor moves to the appropriate data-entry field, allowing reentry. After making the change, the DO YOU HAVE CHANGES? prompt reappears, permitting additional changes, as necessary, until the user is completely satisfied.

One requirement of this type of verification is that each prompt must be numbered so that the user can identify it to make changes. Page verification does not actually require full-screen data entry, although it helps. If full-screen entry is not used, then all of the entries must somehow be displayed simultaneously after they are entered.

How do you write code that permits page verification? After data entry, you must generate a prompt that says "DO YOU HAVE CHANGES? (Y/N)", and, if the user enters Y, another that allows the user to select the item number to change.

Suppose that the user enters a Y and selects an item to change. What then? This is where it gets harder. If the user wants to change something, then the program must represent the data-entry prompt and collect the keystrokes, doing all of the error-testing and other things that went with data entry the first time.

If our data entry program consists of a series of routines, we cannot simply send control back to the one for the variable the user wants to change, or it may require reentry of data that the user does not want to change. For example, the program collects three items of data, and the user wants to change the first one. When we send control back there the user will be forced to reenter all three items before being able to continue the program.

The only solution to his dilemma is to make each of the data-entry routines into a subroutine that operates independently of the others. In this way, if the user wants to change item #1, the program can access the appropriate data-entry subroutine without affecting other data entries.

In short, if we want to do page verification, we must reorganize our program. The reorganization is not complex, but it may require some reorientation. I will show how to reorganize the program presently, but first let us develop a subroutine to handle verification prompting.

Verification prompting can be done with or without a subroutine, although if you do very much of it, a subroutine is a definite help. Even if you prefer not to use a subroutine, the techniques employed in the following verification subroutine will be useful.

Figure 5-16 contains a revised version of the full-screen data-entry program shown earlier in Fig. 5-15. This version has all of the features of the earlier version and, in addition, permits page verification.

Let us begin by examining the Verification subroutine, which consists of lines 3350-3500. (Note that this subroutine calls subroutines 3010 and 3200, and that subroutine 3200 calls subroutine 4100. Thus, all three subroutines—3010, 3200, 4100—must be present in the program.) The arguments of the verification subroutine are V, H, and N. V is the row number and H the column number at which the verification prompts are to be printed. N is the number of verification options available (the number of prompts displayed on the screen). When this subroutine is called, it first presents this prompt:

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
3010 REM--INPUT# DATA ENTRY--
3020 SYS C0,1,V,H :REM CLEAR LINE
3030 SYS C0,0,V,H:REM POSITION PROMPT
3040 PRINT CHR$(18);:REM REVERSE ON
3050 PRINT P$+":";:REM PRINT PROMPT
3060 PRINT CHR$(146);:REM REVERSE OFF
3070 OPEN 1,0
3080 INPUT#1,A$
3090 PRINT
3100 CLOSE 1
3110 RETURN
3200 REM--ERROR MESSAGE--
3210 SYS C0,1,V,H:REM CLEAR LINE
3220 FOR B=1 TO 10
3230 TX=.2:GOSUB 4100:REM .2 SEC DELAY
3240 SYS C0,0,V,H:REM POSITION CURSOR
3250 PRINT CHR$(18)ER$:REM PRINT ERROR MESSAGE IN REVERSE VIDEO
3260 TX=.2:GOSUB 4100:REM .2 SEC DELAY
3270 SYS C0,0,V,H
3280 PRINT CHR$(146)ER$:REM PRINT ERROR MESSAGE IN NORMAL VIDEO
3290 NEXT
3300 SYS C0,1,V,H
3310 RETURN
3350 REM--VERIFICATION PROMPT--
3360 SYS C0,1,V,H:REM CLEAR LINE
3370 SYS C0,0,V,H:REM POSITION PROMPT
3380 PRINT CHR$(18)"DO YOU HAVE CHANGES? (Y/N): ";
3390 POKE 204,0:REM FLASH CURSOR
3400 GET A$:IF A$="" GOTO 3400
3410 IF A$<>"Y" AND A$<>"N" GOTO 3400
3420 IF A$="N" THEN A=0:GOTO 3500:REM NO CHANGES
3430 P$="SELECT ITEM # TO CHANGE"
3440 GOSUB 3010:REM CALL DATA ENTRY SUBROUTINE
3450 A=VAL(A$)
3460 IF A>=1 AND A<=N GOTO 3500
3470 ER$="NUMBER MUST BE BETWEEN 1 AND"+STR$(N):REM DEFINE ERROR MESSAGE
3480 GOSUB 3200:REM CALL ERROR MESSAGE SUBROUTINE
3490 GOTO 3430:REM HAVE USER SELECT ANOTHER ITEM#
3500 RETURN
4100 REM--TX SECOND DELAY--
4110 FOR A=1 TO TX*870
4120 NEXT
4130 RETURN
10000 REM--FULL SCREEN DATA ENTRY PROGRAM--
10010 REM-INITIALIZE SCREEN-
10020 PRINT CHR$(147)
10030 SYS C0,0,4,1
10040 PRINT CHR$(18)"1. NAME:"
10050 SYS C0,0,8,1
10060 PRINT CHR$(18)"2. AGE:"
10070 SYS C0,0,12,1
10080 PRINT CHR$(18)"3. SEX:"
```

```
10090 REM--COLLECT DATA ENTRIES--
10100 GOSUB 10240:REM NAME
10110 GOSUB 10360:REM AGE
10120 GOSUB 10470:REM SEX
10130 REM--VERIFICATION--
10140 V=22:REM VERIFICATION ROW
10150 H=1:REM VERIFICATION COLUMN
10160 N=3:REM NO. PROMPTS
10170 GOSUB 3350:REM CALL VERIFICATION SUBROUTINE
10180 IF A=0 GOTO 10220:REM ALL ENTRIES OKAY
10190 IF A=1 THEN GOSUB 10240:GOTO 10140:REM CHANGE NAME
10200 IF A=2 THEN GOSUB 10360:GOTO 10140:REM CHANGE AGE
10210 IF A=3 THEN GOSUB 10470:GOTO 10140:REM CHANGE SEX
10220 END:REM ALL ENTRIES VERIFIED
10230 REM--DATA-ENTRY SUBROUTINES--
10240 REM-NAME-
10250 V=4
10260 H=1
10270 P$="1. NAME"
10280 GOSUB 3010
10290 NAME$=A$
10300 REM-ERROR TEST-
10310 IF LEN(NAME$)>=2 AND LEN(NAME$)<=24 GOTO 10350
10320 ER$="NAME MUST BE 2-24 CHARACTERS LONG"
10330 GOSUB 3200
10340 GOTO 10270
10350 RETURN
10360 REM-AGE-
10370 V=8
10380 P$="2. AGE"
10390 GOSUB 3010
10400 AGE=VAL(A$)
10410 REM-ERROR TEST-
10420 IF AGE>=2 AND AGE <=99 GOTO 10460
10430 ER$="AGE MUST BE BETWEEN 2 AND 99 YEARS"
10440 GOSUB 3200
10450 GOTO 10390
10460 RETURN
10470 REM-SEX-
10480 V=12
10490 P$="3. SEX"
10500 GOSUB 3010
10510 SEX$=A$
10520 REM-ERROR TEST-
10530 IF SEX$="M" OR SEX$="F" GOTO 10570
10540 ER$="PLEASE ENTER 'M' OR 'F'"
10550 GOSUB 3200
10560 GOTO 10500
10570 RETURN
```

Fig. 5-16. Full-screen data-entry program based on Fig. 5-15, but with the addition of data-entry verification.

DO YOU HAVE CHANGES?(Y/N)

If the user types in N, then the variable A is set equal to zero and control RETURNs to the program. If the user types in Y, then A is set equal to the number of the item to change. Thus A carries back with it the message of change or no change to the displayed item.

Now let us see how this subroutine works. Lines 3360 and 3370 clear the line and position the cursor. Line 3380 PRINTs the first verificaton prompt in reverse video. Line 3390 displays a flashing cursor. Line 3400 GETs a single character, and line 3410 tests it to make sure that it is either Y or N, recycling control to line 3400 if it is not.

Line 3420 tests whether the entry is N. If so, A is set equal to 0 and control is sent to line 3500, which RETURNs to the program, thereby ending the subroutine.

If the user types in Y, then lines 3430-3490 are executed. Lines 3430 defines a prompt, P$, which is displayed when line 3440 calls the Data-Entry subroutine at line 3010. Line 3450 evaluates the variable returned from the Data-Entry subroutine and line 3460 conducts a range test, sending control to line 3500 if the test is passed. If the test is failed, then the user has typed in an invalid ITEM# to change, and line 3470 defines an error message, which is displayed when the Error-Message subroutine is called at line 3480. Line 3490 then sends control back to line 3430 so that the user can type in a different ITEM#.

Now let us consider how the actual program works, starting with the data-entry subroutines. The data-entry program consists of lines 10000-10570. There are three data-entry subroutines, starting at lines 10240, 10360, and 10470. Each of these has the same structure, so let us focus on the one starting at line 10240, which collects the NAME$ variable. This subroutine consists of lines 10240-10350. It should look very familiar to you, since it is almost identical to lines 10100-10155 of Fig. 5-15, which we discussed earlier in this chapter. Other than line numbers, only two changes have been made: the prompt is now preceded by the number "1," and a RETURN statement has been added at the end. Analogous changes have been made to the routines for collecting the AGE and SEX$ variables.

Now let us examine the control code, which consists of lines 10000-10220. Lines 10000-10080 create the data-entry screen. These lines are identical to the corresponding lines in Fig. 5-15.

Line 10100 calls the first data-entry subroutine at line 10240. Lines 10110 and 10120 call the other two data-entry subroutines.

The verification part of the program begins at line 10130 and continues to the end of the control code. Lines 10140, 10150, and 10160 set the arguments for the verification subroutines. These arguments will cause the verification prompt to appear at row 22 and column 1 of the display, and to accept any "ITEM NUMBER TO CHANGE" between 1 and 3. The Verification subroutine is called at line 10170. If the value of A RETURNed by the subroutine is 0, then there are no changes to make, and line 10180 sends control to line 10220, terminating the program.

If the value of A RETURNed by the Verification subroutine is 1, 2, or 3, then one of the lines 10190-10220 will execute, calling the appropriate data-entry subroutine, and then sending control back to line 10140, where the user can reverify and select further changes to make, if necessary.

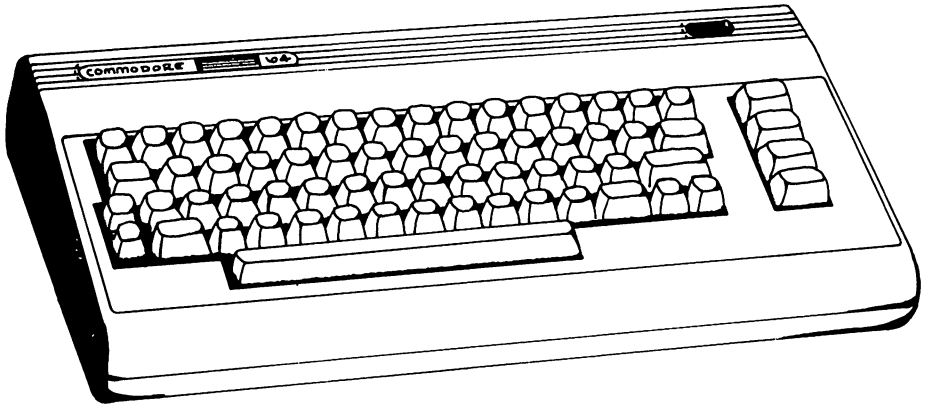
This program may seem a bit convoluted to you if you are not used to constructing programs out of building blocks such as subroutines. If you have more experience, you may still find it difficult to follow. On the other hand, it may all be perfectly clear to you.

Whatever your state of mind, spend a little time going over the code and the explanation just given. This is an example of how your control code shrinks when you construct a program of powerful subroutines. Note that, although there are many lines of code in this program, the controlling part consists of only lines 10090 through 10210—a total of 14 lines, two of which are REMs and one of which is END. In other words, the intelligence of this program is contained in only 11 lines, which do their job mainly by calling on their friends, the subroutines. When you look at the program from this perspective, you see that it is much simpler than it could be without subroutines. If you are not convinced of this, consider how you might write a program that does what this one does without them.

Figure 5-16 illustrates the basic structure and logic for doing page verification of prior data entries. This technique is also useful in programs that make use of a data base in which users need to

interrogate and modify the data base. The programming requirements for a data-base editing program are much the same as those for page verification. The main difference is how long after the entries are made they are reviewed and edited. In a verification routine, this is a few seconds or minutes. In a data-base editing program, it may be days, weeks, or months afterward.

Chapter 6



Program Control

To make a computer program do something, the user must exercise control over it. In a game, this may be done with a joystick which, for example, tells the gunnery platform where to move and when to fire at the descending invaders. In a more serious program, control is usually exercised through the keyboard. A popular method of control in microcomputer programs is the menu. This is simply a list of the functions that the program can perform. The user looks at the menu to find the function desired, and then uses the keyboard to type in the selection with a number, letter, or function key. The computer then, like an obedient servant, carries out its master's command. Although the menu is popular, other control methods may be used in addition to or in place of it. This chapter describes the control methods most popular with microcomputers—menus, simple choices, and typed-in commands.

To use any control method, the user must enter information into the computer. Since entries are made, users may make errors, and all of the error-testing methods described in Chapter 5 apply. If you are not familiar with these methods or have not read Chapter 5, do so before starting this chapter. Error tests are included in the code used in this chapter, but the discussion does not dwell on them since they were covered in Chapter 5.

Data entry and program control are more similar than different. Both involve user entry of data which are assigned to variables. Error tests and user verification of entries are required in both cases. There is, however, at least one big difference. The entries made during a data-entry program become part of the program's data base and affect the content of the displays the user views. The entries made for program control govern what the program does. They decide, for example, whether the user views an analysis display, updates files, or exits the program. If you think of a program as a sort of network of roads, with each road leading to a different program function, then the control entries decide what routes are taken. To extend the metaphor, conventional data entry decides what data are processed—who gets in and out of the vehicles, and when and where they do it.

The method of control has a big effect on how easy it is to learn and use a program. Several trade-offs must be made in selecting the control method. Making these trade-offs is not always easy. The three primary trade-off factors are ease of learning, ease of use, and speed.

Ease of learning is always a desirable goal, despite what some misguided teachers and writers may think. No virtue accrues to the person who learns things the hard way, and in fact it is more the other way around. What makes a control method relatively easier or more difficult to learn? Several things matter, but one of the main ones is how much the user must commit to memory. Since menus display what is available, the user does not have to memorize their content, and this makes them relatively easy to learn. If the program's functions are not displayed on a menu, then the user must memorize what is available and create an internal list of program options. This is harder to learn, obviously, but not necessarily a bad thing. Another thing that influences ease of learning is the logic (or lack thereof) that holds the program's control structure together. Is that structure a nice, geometric, spider's web, or a rat's nest of confusion?

Some control methods are easier to use than others. Also, given a particular method, the way you implement it affects how difficult it is to use. A complicating factor is that making something easy to learn often makes it more difficult to use, and vice versa. For example, providing on-screen prompting makes a program easy to learn. After a user masters a program, however, such prompting often gets in the way, making the program more difficult to use.

Speed is important, and in some programs it is more important than ease of learning or use. Some control methods are faster than others, but the fastest control method is not always the best.

Often, you must make a trade-off among making your program easy to learn, easy to use, or fast. In general, programs with menus are the easiest to learn and use, but they are also among the slowest. It seems—on the surface at least—that you cannot have everything. As this chapter will show, it is not really as bad as it might seem, because there are ways to combine the control methods to get the best of both worlds. Before you can do this intelligently, however, you must understand each control method and its strengths and weaknesses in terms of the three trade-off factors.

The chapter begins by presenting design guidelines for program control, and then covers the three common control methods. Although all three control methods are covered, the main emphasis is on menus. Menus are not necessarily the answer to every programmer's design problem, but they are becoming increasingly popular and widely used in microcomputer programs for very good reasons. The final section shows how to combine menus and typed-in commands to get the best of at least two worlds. As in the earlier chapters, this one presents many examples of code and more than a few useful subroutines.

PROGRAM CONTROL DESIGN GUIDELINES

Guess which two design principles are (again) important for designing program control. As always, simplicity and consistency are the ones.

Start with simplicity. Recall the analogy I made between program control and a road network. In this analogy, the various roads are the links between the functions the program can perform (its subprograms), and the crossroads are where control actions occur. This network is often referred to as a control structure. Well, the more complex this network becomes, the more difficult it is for someone to master the program. Have you ever been to a city where the roads were poorly laid out, or the maps were bad? You know, lots of dead ends, roads that ran off at odd angles, roundabouts, and other nonstandard interconnections that broke up the regularity of the pattern. If you live in a city that has such roads, eventually you get used to them. If, however, you are a stranger who is used to a more regular pattern—one that is simple and that follows conventional rules for how streets should interconnect—then finding your way around can be very difficult.

When you move to a city with such roads, you must go through a learning phase. At first it all

seems very confusing, but with a map and time, you master the roads. Eventually, you may discover that there was method to the madness of the road patterns after all. That is, some of the crazy interconnections, odd angles of roads, and so forth, make it possible to get between points more quickly. After learning the road system, you may grow to like it. It may not be good for strangers, you realize, but it is fine for old hands who live there, such as yourself.

It may not be obvious, but there is a very close analogy between this example and the control structure you build into your program. A simple control structure is easy to learn and use at first, but it may be more difficult to use later. A more complex control structure may be faster once you have mastered it, but it may be harder to learn.

In microcomputer programs, it is best to keep things simple, and to lean in the direction of making your program easy to learn. I will show some concrete examples of how to do this in my discussion of menu networks, later in the chapter.

Another way simplicity applies is in what you name things, that is, your use of language. Find the simplest, most descriptive labels that you can. This applies universally to anything in a computer program that needs a label—the name of a program, a display's title, or a procedure that must be performed. Avoid the complex, technical, computer-sounding, and abstract. Instead, use simple, concrete, descriptive terms. To illustrate, if you just LOADed up a new program for the first time, which menu do you think would be the easiest to understand, that in Fig. 6-1 or that in Fig. 6-2? Keep it simple.

Now take consistency. Following the consistency rule makes it easier for someone to learn something because it reduces the number of different things that must be learned. This rule requires you to do similar things in similar ways. How does it apply to program control?

Again, it is helpful to answer this with a con-

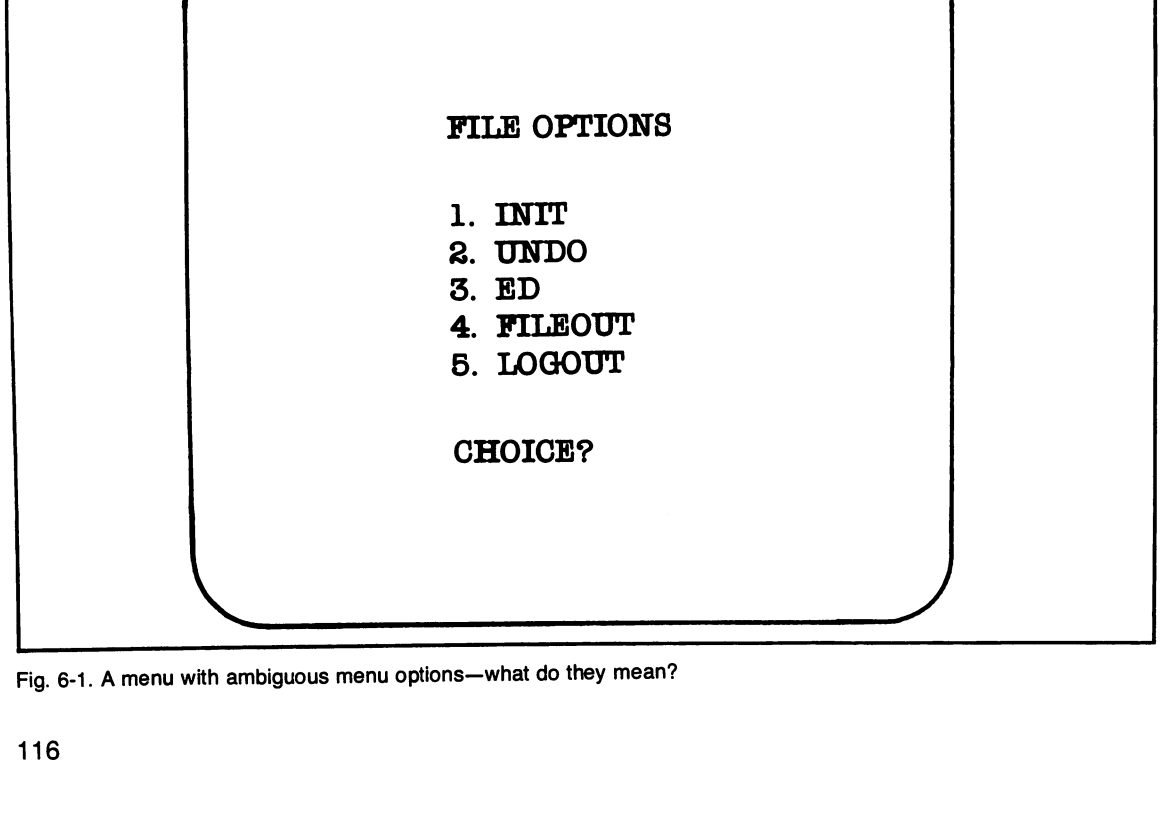


Fig. 6-1. A menu with ambiguous menu options—what do they mean?

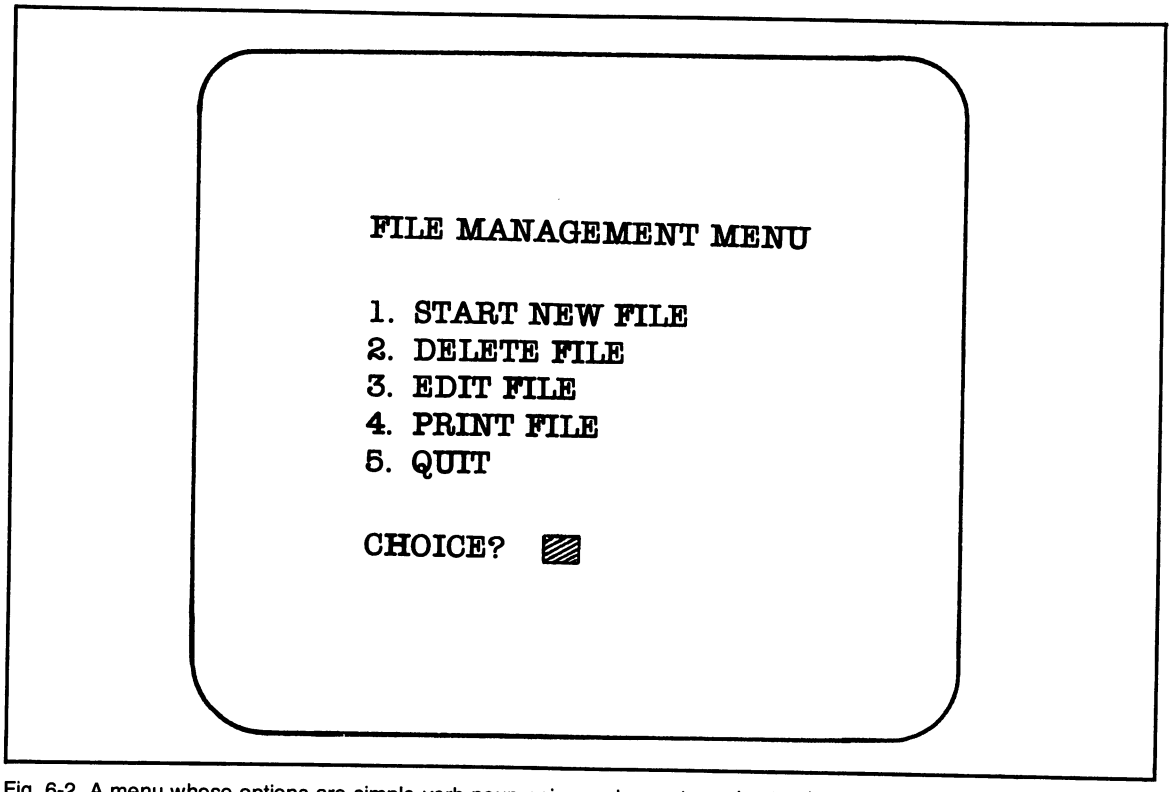


Fig. 6-2. A menu whose options are simple verb-noun pairs, and easy to understand.

crete example. Suppose that you try out a new program that uses menus. The first menu you encounter looks like that shown in Fig. 6-2. Some features of this menu are:

- It has a title with the word "menu" in it.
- Menu options are numbered.
- The options are centered both vertically and horizontally.
- The menu options consist of verb-noun pairs.
- A prompt directs the user to type in CHOICE.
- A flashing cursor follows the prompt.
- The last menu option is labeled "Quit."

Now, suppose that you select an option from this menu, and the program displays a screen that looks like that shown in Fig. 6-3. What is this? In some ways it looks like a menu, but:

- The word "menu" does not appear in the title.
- The items are left-justified.
- The options (if that is what they are) are not numbered, but they are preceded by capital letters.
- The options consist of nouns.
- There is no prompt or cursor.
- There is no "QUIT" option, but the last option is labeled "Escape."

What do you make of this? It is probably not difficult to figure out that this also is a menu, but it is not like the first one. Since its format differs, it poses a certain amount of ambiguity to the user. Making sense of it is a problem-solving task. First, the user must figure out what the screen is. Like the scientist, he or she develops a hypothesis. The user thinks something such as, "I think that that is a menu. Let's see, now. Menus have properties A, B,

and C. Let me test my hypothesis by examining the content of this screen. No, it does not have property A. I think it has property B. I am not sure about property C. . .".

You get the idea. Why make it necessary for the user to act the role of a scientist? Make it easy on the user. Be consistent.

MENUS

I defined a menu in general terms, but for the record (and the sake of consistency) I will define it a little more formally. A *menu* consists of a list of program options and a data-entry routine that allows the user to select one of the options. This is the minimum, of course. A menu can, and usually should, include more than just these two things.

What Is Good about Menus

Software publishers, in advertising their programs, often mention that they are "user-friendly" and have menus. Many of the programs whose listings appear in books and magazines use menus. There must be something good about them, or else they would not be so widely used. Well, what is it?

Perhaps the best way of answering this question is to consider the alternative. The alternative to a menu is a blank screen. This, in fact, is all that many programs—especially advanced ones—provide. To use a program such as this, the user must pull the program options out of his or her memory.

A program menu is, well, a menu. It is aptly named. Walk into most restaurants and they hand you a menu or have a menu posted. Walk into a fast-food place and the menu is simple and easy to figure out. Walk into a Chinese restaurant and it may take you hours. Walk into a fancy restaurant, or one that has such pretensions, and the menu is not written down, but related verbally by a fancily dressed maitre d' with a real or phony foreign ac-

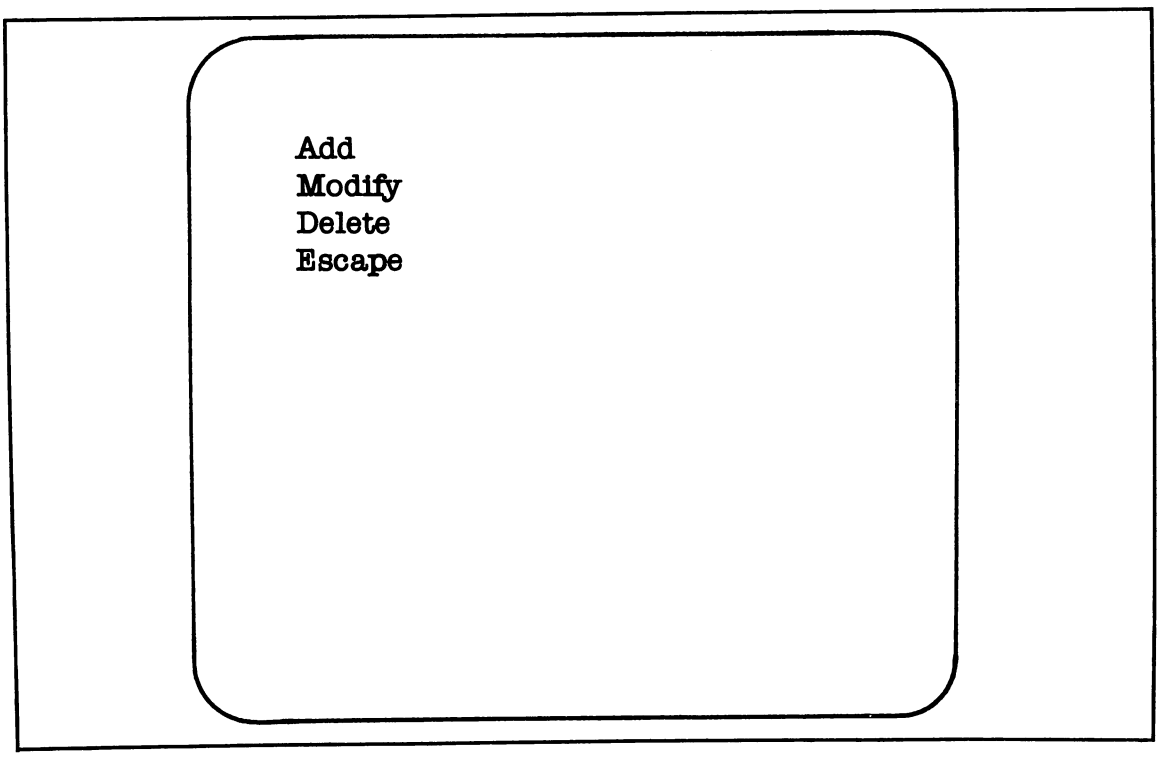


Fig. 6-3. A menu is not always obviously a menu. This one lacks a title and prompt line, and it is not clear how options are to be selected.

cent. However the menu is delivered to you, it fulfills a need you have to know what is available. Without it you could not order without already knowing what was available or asking someone. In a computer program, you do not usually have the second option.

What Is Bad about Menus

Some people do not like menus. Many sophisticated users regard them with about the same amount of enthusiasm as a 4-minute miler would regard training with a high school track team. The problem for these experts is that they do know the bill of fare and do not need a menu to tell them what they can order. When the waiter hands it to them, they hand it right back, and order on the spot. They do not like delays. They want to get on with things. Some programs have many menus, and to get to the program you want, you may have to work your way there, menu by menu. This can be time consuming and tedious for the expert.

So, there is a good side and a bad side to using menus. They are not the answer for every program; however, they are the answer for many.

FULL-SCREEN AND PARTIAL-SCREEN MENUS

Menus can be designed in a number of different ways. This section describes two types: (1) full-screen menu, and (2) partial-screen menu. A full-screen menu is a menu of the type illustrated in Figs. 6-1 through 6-3. Full-screen menus are typically used as main control menus. When they are on the display, nothing else appears. A partial-screen menu is typically displayed on one part of the display while something else is on the rest of the display. Such a menu is often used to control the information on the display—for example, it might be used to modify the scale or other graphic features of the display.

Both types of menus are useful, but usually for different purposes. Let us start with the full-screen menu.

Full-Screen Menu

Figure 6-2 is a typical full-screen menu. The main features of this menu are as follows:

- Menu title—This is centered at the top of the display. It is in all capital letters, reverse video, and contains the word "menu."
- Menu options—This menu contains five options, numbered 1 through 5.
- Prompt line—This is the line on which the user types in the menu choice.

As mentioned earlier, a menu may contain more than just these three features. In a moment, I will discuss what these might be, but let us start with the main features and see why these three simple features are not as simple as they seem.

Title. The title, of course, is straightforward. I suggest centering it at the top of the display, or perhaps a few lines down from the top, if you can compose a better-looking display that way. I recommend the use of all capital letters. The use of reverse video is optional, somewhat a matter of style. In this book, the style followed is to use all capital letters and reverse video for titles, headings and prompts. While following this convention is not critical, it is important to follow a convention that permits the user to discriminate between titles, headings, and prompts, and the other information on the display.

Options. Now let us consider the menu options. There are a number of dos and don'ts here. First, the way the options are identified is somewhat arbitrary. In Fig. 6-2, each option has a number beside it. Options may also be identified with letters (Fig. 6-4). A third way, popular with the C-64, is to list them with function key assignments (Fig. 6-5). The method I prefer and follow in this chapter is to use numbers. Again, there is nothing magic about them, and this is mainly a matter of style or preference.

When numbers are used, the lowest numbered option should be the number 1. You often see a menu with an option numbered zero on it. Avoid this. These numbers stand for real things—programs or program options. A zero implies the absence of something, and is an abstract notion at

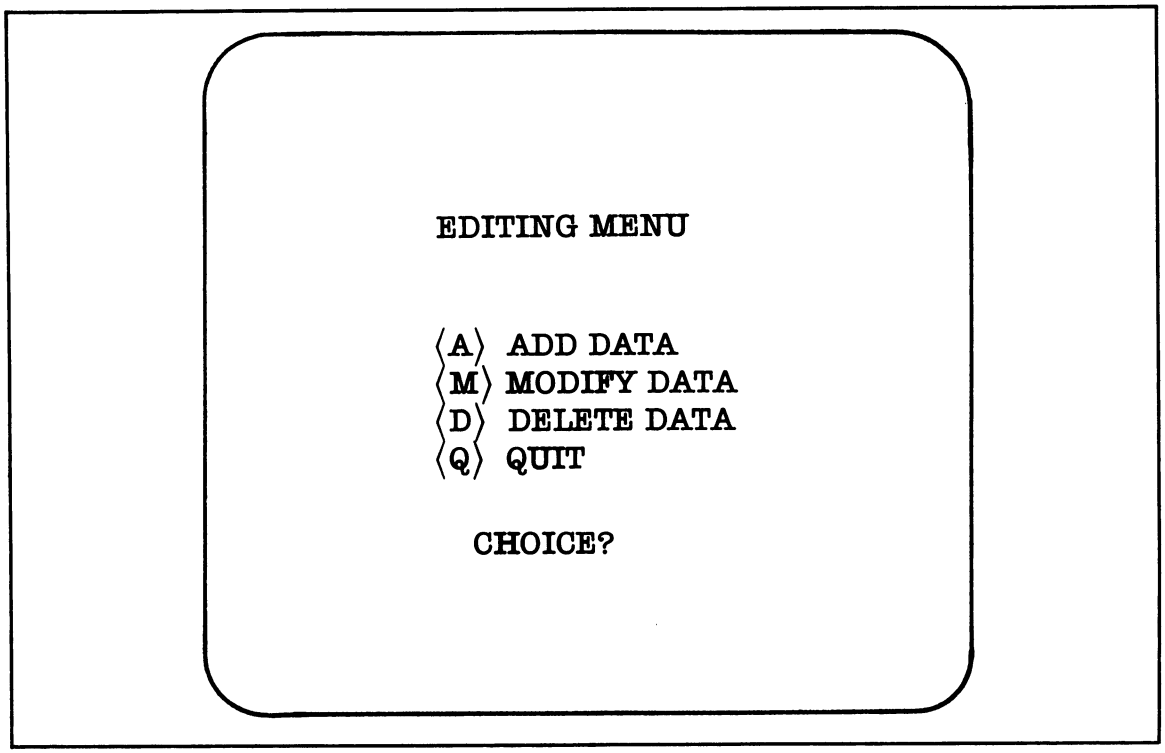


Fig. 6-4. A menu whose options are selected with a letter.

best. It took the human race a long time to discover the concept of zero, and many of us still have difficulty with it. Please help us out by not using zeros in your menus.

Using letters as option identifiers is good if your program has a few menus, and you can assign letters that relate mnemonically to the options. For example, let the user type in "A" to Add something, "M" to Modify something, and so forth. Users tend to expect that the identifiers relate semantically to the options. If they lack a semantic component (that is, an intrinsic meaning), then you are better off using numbers.

Now we come to function keys. Your C-64 has them, and they are nice. What bothers me about them, frankly, is that, while there are eight function key assignments, there are really only four function keys. If you look at your C-64, you will see that these are labeled F1, F3, F5, and F7. You can activate F2, F4, F6, and F8 by pressing the shift key together with the F1 through F7 key, respectively. There are some problems here. First, it is desirable to allow users to select their menu option with as few keystrokes as possible (the laziness principle). The requirement to use the shift key for even-numbered function key selections goes against this principle. Second, although we can use four of the function keys without doing this, they are odd-numbered. We cannot in good conscience, create a menu with options F1, F3, F5, and F7. "What happened to the missing options?" the user will ask. If we list options from, say F1 through F6, then a different problem arises. Now it is possible to select options 1, 3, and 5 with a single keystroke, but selecting options 2, 4, or 6 takes two. This violates the consistency principle.

No menu should have more than about nine options. The ideal number is somewhere in the five to seven range. This is not a number that I dreamed up, but one that has been established through research. When a menu is longer than about nine items, it becomes more like a directory. It takes a

fairly long time to read such a menu. Have you ever used a program with a menu that had 20 or 30 options on it? Creating a menu this long shows a lack of awareness that it takes time for people to search for and read things.

If you create a menu with more than five options, it is a good idea to separate the list of options into two parts, separated by a space (Fig. 6-6). This makes it easier for the user to read.

In general, it is desirable for menu options to be concise and descriptive. If the user rarely uses the program, then it makes sense for the options to be longer and more descriptive. Short options can be read more quickly, and save the experienced user time. It is hard to make short options as descriptive as longer ones, however, which argues for longer options on menus that are rarely used.

Menu options can take on several different forms, such as:

- Short action-object. Examples are UPDATE FILES, EXIT PROGRAM, and START NEW FILE. These define an action and something to be acted upon by the user.
- Short descriptive label. Examples are FILE UPDATE, EXIT, and NEW FILE. These labels may be the names of programs or define an implicit action and object. This form of label is less explicit than the action-object form.
- Long descriptive label. Sometimes these are framed as questions, such as DO YOU WANT TO UPDATE YOUR FILES? Alternatively, they may just describe an action: START A NEW FILE.

Options can be constructed in other ways, although these are the common ones. It is difficult to say which is best, since this depends upon the program and the user. It is clear that an option form that gives an action and something to be acted upon

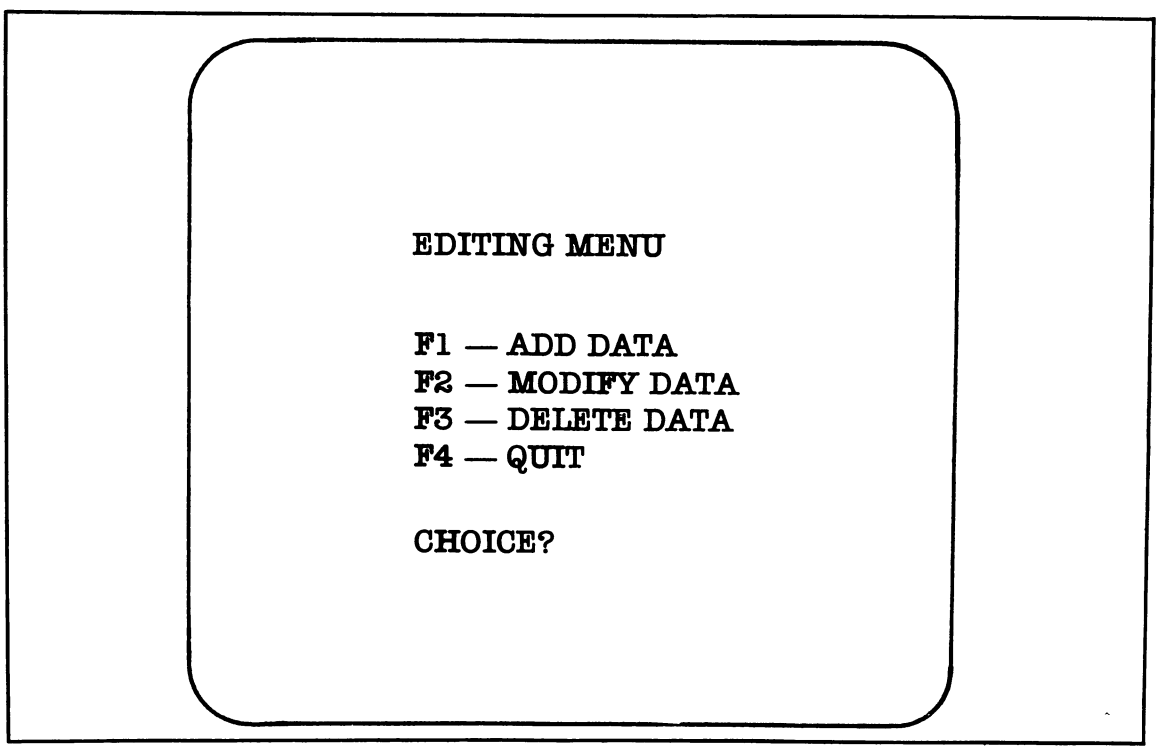


Fig. 6-5. A menu whose options are selected with function keys.

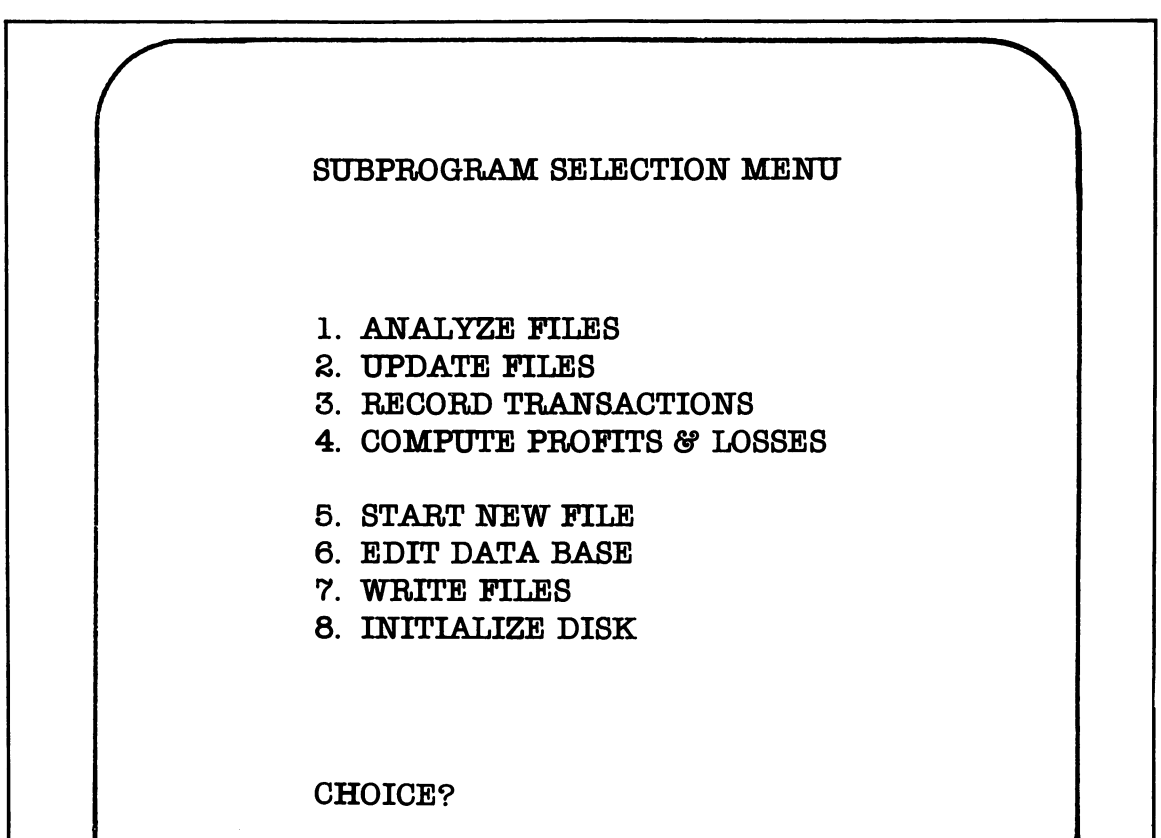


Fig. 6-6. A long menu, whose option block is broken into two parts to make it easier to read.

is more meaningful than a descriptive label, especially for the inexperienced user. Long labels are also better for the inexperienced user. Longer labels take more time to read, however, and may be less desirable in programs that are frequently used.

I prefer the short action-object form since it is short and also reasonably descriptive. Whatever form is used, it is important to avoid mixing options in different forms on the same menu or within the same program. Doing so can cause confusions. For example, if most of your menu options take the short descriptive label form, a user may be confused by an option with the label UPDATE FILE. Does this refer to a file labeled UPDATE or to the act of updating any file?

In programs that involve many menus, linked together in a network, it is often desirable to make the final option of all menus the same, and to use this option for exiting the current menu and returning to the previous menu. To illustrate, Fig. 6-7 shows a family of menus that are linked together in a hierarchical fashion. The last option of each menu permits the user to exit from that menu and return to the previous one. In this case, the last option on every menu is called by pressing the Function 1 (F1) key. A user who is "deep" in the program does not have to look at the menu to know how to back up to a higher-level menu—pressing the F1 key always does this, and pressing it repeatedly brings back the highest-level menu.

The Prompt. The prompt for a menu is like a data-entry prompt in that it must (1) alert the user that an entry is required (flashing cursor does this), and (2) describe the nature of the entry. Prompts tend to be written in either succinct or descriptive form, depending upon the proclivities and moods of programmers, and either form is perfectly acceptable. For a descriptive prompt, you might use something such as:

- PLEASE TYPE IN THE NUMBER (OR LETTER) OF YOUR CHOICE
- PLEASE TYPE IN YOUR CHOICE
- TYPE IN THE NUMBER OF YOUR PROGRAM SELECTION

Some programmers use the term "enter" instead of "type in" or "please type in". "Enter" is, however, a bit ambiguous. "Type in" is more specific, descriptive, and accurate in terms of what the user must actually do.

I prefer short prompts—something along these lines:

- CHOICE
- OPTION
- SELECTION NUMBER

While it is true that these prompts are not particularly descriptive, the form of the menu is such that there is little doubt that what is being viewed is a menu and that what the user must do is type something in at the keyboard. After using menus a few times, users tend to ignore the prompt line, anyway.

When the user types in a menu selection, the typed entry should appear (that is, be "echoed") on the prompt line after the prompt. The program should not execute the choice immediately, but should give the user a chance to view the selection and change it, if desired. In other words, the user should be able to verify the entry.

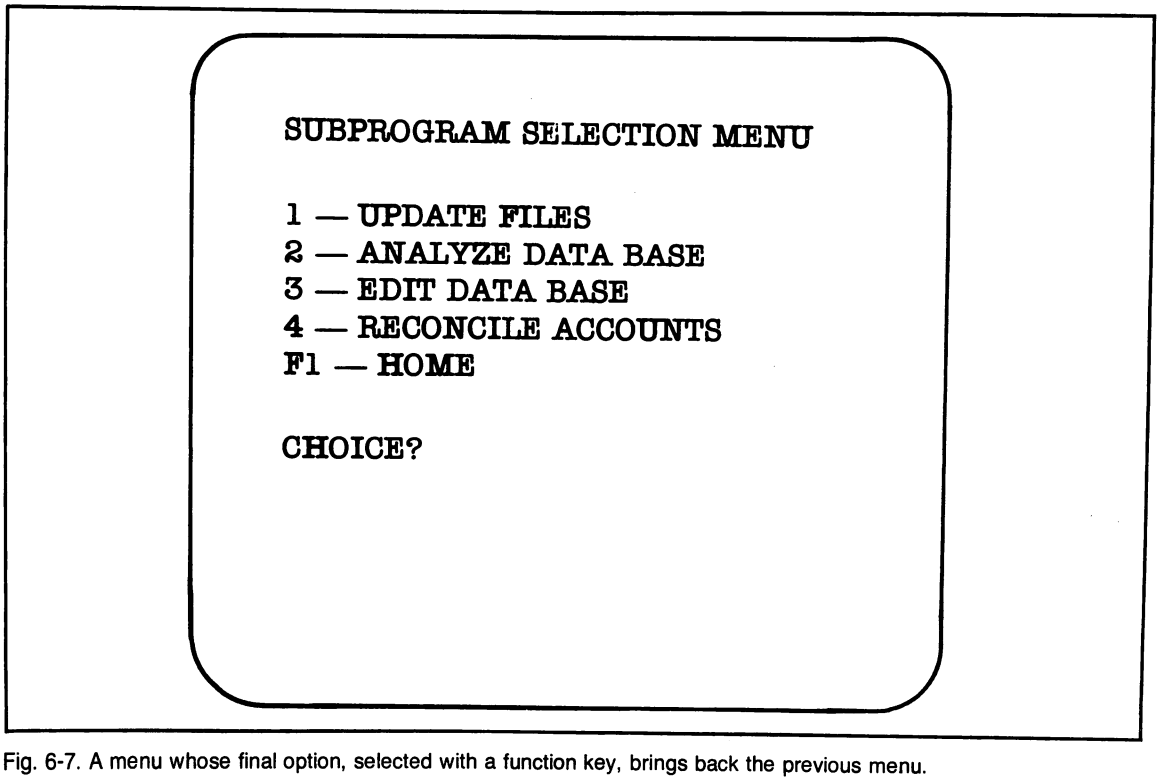


Fig. 6-7. A menu whose final option, selected with a function key, brings back the previous menu.

The simplest way to permit user verification of a typed-in entry is to use the INPUT statement. Using this statement, whatever the user types in is echoed to the screen, and nothing takes effect until the Return key is pressed. As noted in Chapter 5, the INPUT statement has a drawback in that it allows a careless or mischievous user to wreak havoc on a screen display. For this reason, it is better to use a Return-verified GETting technique similar to that described in Chapter 5. By using such a technique, you can maintain control of the screen and also filter the keyboard entries. I will show how to do this later in this chapter.

Additional Information. Depending upon the nature of the program, it sometimes makes good sense to display certain additional information on the menu screen. This is particularly true if the program tends to use a lot of menus, and the user frequently has one on the display. In this case, the menu is a convenient place to present status or file information. The information should only be presented on the menu if it is of genuine importance to the user. Otherwise, it is clutter and a distraction from the central purpose of the menu screen.

Here are examples of types of information that might be presented on a menu screen:

- The amount of available memory remaining.
- Dates—current date, date file last updated.
- The name of the file presently active.
- The amount of available file space remaining.

One way to add such information is to put it on the bottom of the menu, below the prompt line. This way the information is accessible, but out of the way. The user can refer to it, if desired.

Generating a Full-Screen Menu

There are two common ways to generate a menu screen:

- Directly—Like a display screeen.
- With a subroutine—That is, supply certain arguments, and have a subroutine create the menu from them.

Generating a Menu Directly. If your program uses only one menu, or only a few, then it may be best to generate directly. Lay out the menu on paper, just as you would any other display screen. Then write the code to create the menu.

To illustrate, Fig. 6-8 is the design matrix for a simple program control menu. Figure 6-9 contains the code for generating this menu, line by line. The menu was laid out in such a manner that the block of options is centered vertically; that is, it is equidistant from the top and bottom of the screen. It is also centered horizontally.

Since the options vary in length, it is a little tricky to center the option block horizontally. Centering is done by computing the average length of an option, subtracting this from 39, and dividing by 2. (This is the same technique used for centering a character string on the screen, as in subroutine 1500.) After computing the horizontal TAB position in this manner, lay out the options on the matrix and see how they look. If you have some very short and very long lines, the option block may tend to look off center, because we tend to pay more attention to long lines than short ones. See how the option block looks to the eye. Then readjust the horizontal TAB until it looks good. This is a subjective, aesthetic judgment. (I apologize for not having a subroutine to do this for you.)

The title is at the top of the display. You might move it down a few lines if it looks better to you that way. This is a matter of personal preference. The prompt is at the bottom of the display. Again, move it up a few lines if you like.

If you decide to add additional information below the prompt line, you may want to move everything up, until the display appears balanced. It probably will not look right the first time you lay it out.

The code for generating the menu is shown in Fig. 6-9. Line 10010 clears the display. Line 10020 defines the menu title, and line 10030 calls the Center & Print subroutine at line 1500. Line 10050 moves the cursor down to row 10, and lines 10060-

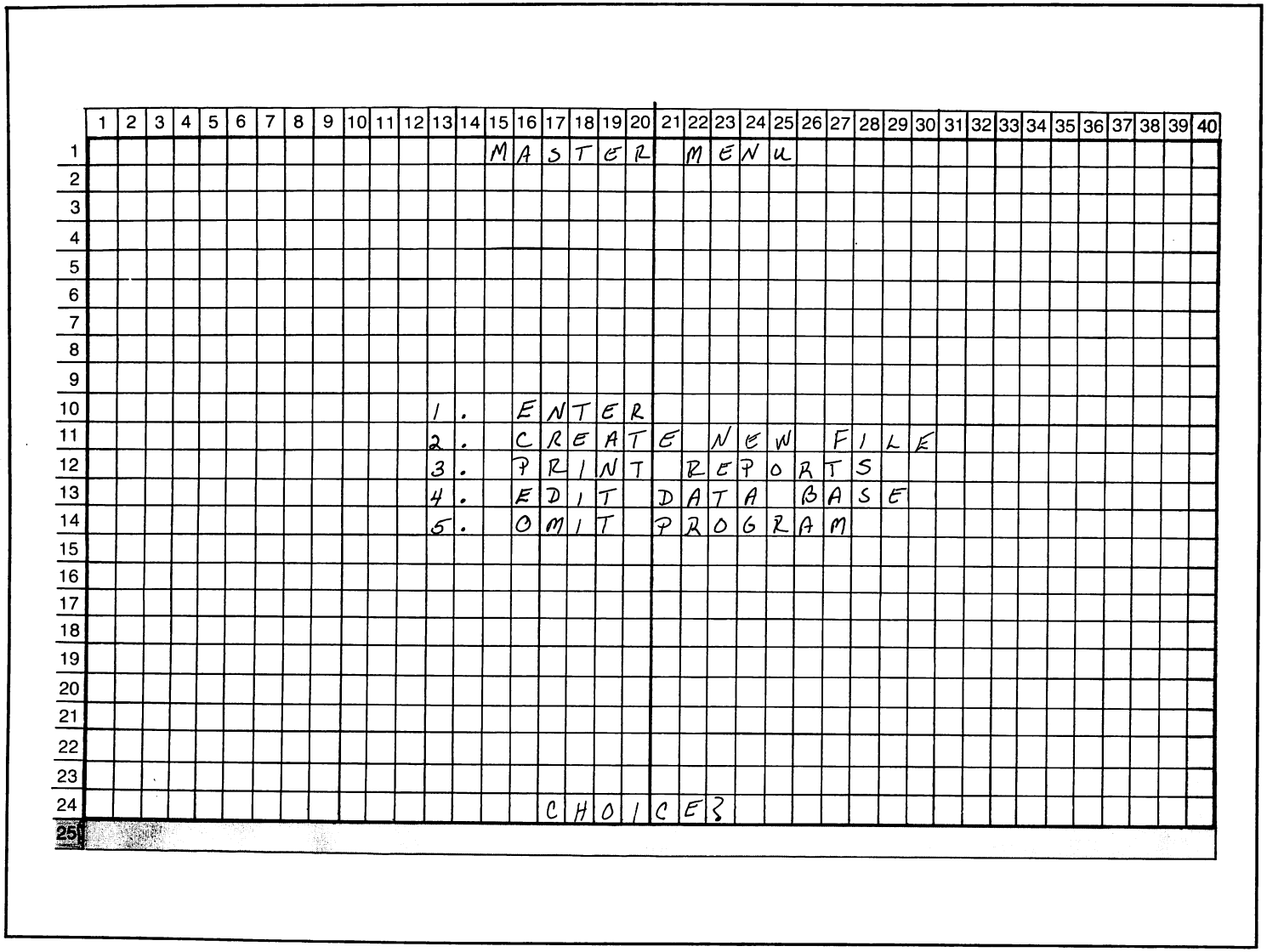


Fig. 6-8. Design matrix for a simple program control menu.

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
1500 REM--CENTER & PRINT T$--
1510 T=(39-LEN(T$))/2
1520 PRINT CHR$(18);
1530 PRINT TAB(T)T$
1540 RETURN
10000 REM--FULL-SCREEN MENU GENERATOR--
10010 PRINT CHR$(147);:REM CLEAR SCREEN
10020 T$="MASTER MENU"
10030 GOSUB 1500:REM PRINT TITLE
10040 REM-DISPLAY OPTIONS-
10050 SYS C0,0,10:REM POSITION FIRST OPTION TO ROW 10
10060 PRINT TAB(12)"1. ENTER DATA"
10070 PRINT TAB(12)"2. CREATE NEW FILE"
10080 PRINT TAB(12)"3. PRINT REPORTS"
10090 PRINT TAB(12)"4. EDIT DATA BASE"
10100 PRINT TAB(12)"5. QUIT PROGRAM"
10110 REM-PRINT PROMPT-
10120 SYS C0,1,24,17:REM CLEAR PROMPT LINE
10130 SYS C0,0,24,17:REM POSITION PROMPT
10140 PRINT CHR$(18)"CHOICE?";
10150 POKE 204,0:REM FLASH CURSOR
10160 GET A$:IF A$="" GOTO 10160:REM GET MENU SELECTION
10170 A=VAL(A$)
10180 IF A<1 OR A>5 GOTO 10120
10190 PRINT CHR$(146)" ";A$;:REM DISPLAY KEYSTROKE
10200 GET B$:IF B$="" GOTO 10200:REM ENTRY VERIFICATION
10210 IF ASC(B$)<>13 GOTO 10120:REM VERIFICATION RETURN KEY?
10220 END
```

Fig. 6-9. Program code for generating the menu shown in Fig. 6-8 directly.

10100 PRINT the five menu options at TAB 12.

Lines 10120-10210 PRINT the prompt and collect user entries. Line 10120 clears the prompt line; 10130 positions the cursor, and 10140 PRINTS the prompt CHOICE? in reverse video. Line 10160 GETs a single keystroke and assigns it to A$. Line 10170 eVALuates the keystroke, and line 10180 conducts a range test. If the number is between 1 and 5, it is valid. If not, control is sent back to line 10120. Note that, unless the entry falls within the valid range—1-5—the user is required to continue re-entering it. Invalid entries are filtered out.

Once a valid entry occurs, line 10190 PRINTs it. Line 10200 then GETs B$, the verification keystroke. This must be a Return keystroke, or control is sent back to line 10120 again, starting data entry once more.

Generating a Menu with a Subroutine. Laying out a menu and then writing the code for it becomes a bit tedious if you have more than one or two of them to do. You can save yourself a lot of time by using a subroutine to do the work for you. The trickiest part is to lay out and then display the block of options. This must be centered both vertically and horizontally. What is involved in figuring out the vertical and horizontal positions can be reduced to simple mathematics of the sort that your

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
1500 REM--CENTER & PRINT T$--
1510 T=(39-LEN(T$))/2
1520 PRINT CHR$(18);
1530 PRINT TAB(T)T$
1540 RETURN
4200 REM--FULL-SCREEN MENU GENERATOR--
4210 PRINT CHR$(147);
4220 T$=T$+" MENU"
4230 GOSUB 1500:REM PRINT TITLE
4240 REM-COMPUTE MENU STARTING COLUMN-
4250 L=0
4260 FOR A=1 TO N
4270 L=L+LEN(OPT$(A))+3
4280 NEXT
4290 L=L/N
4300 H=(39-L)/2:REM COMPUTE TAB
4310 REM-COMPUTE VERTICAL OFFSET-
4320 V=INT(25-N)/2
4330 REM-DISPLAY OPTIONS-
4340 SYS C0,0,V:REM POSITION FIRST OPTION
4350 FOR A=1 TO N
4360 PRINT TAB(H-1)STR$(A)". "OPT$(A)
4370 NEXT
4380 REM-PRINT PROMPT-
4390 SYS C0,1,24,17:REM CLEAR PROMPT LINE
4400 SYS C0,0,24,17:REM POSITION PROMPT
4410 PRINT CHR$(18)"CHOICE?";
4420 POKE 204,0:REM FLASH CURSOR
4430 GET A$:IF A$="" GOTO 4430:REM GET MENU SELECTION
4435 POKE 204,1
4440 A=VAL(A$)
4450 IF A<1 OR A>N GOTO 4390
4460 PRINT CHR$(146)" ";A$;:REM DISPLAY KEYSTROKE
4470 GET B$:IFB$="" GOTO 4470:REM GET ENTRY VERIFICATION
4480 IF ASC(B$)<>13 GOTO 4390:REM VERIFICATION RETURN KEY?
4490 RETURN
10000 N=5
10010 T$="MASTER"
10020 OPT$(1)="ENTER DATA"
10030 OPT$(2)="CREATE NEW FILE"
10040 OPT$(3)="PRINT REPORTS"
10050 OPT$(4)="EDIT DATA BASE"
10060 OPT$(5)="QUIT PROGRAM"
10070 GOSUB 4200
10080 END
```

Fig. 6-10. A subroutine (lines 4200-4490) for a generating full-screen menu with numbered options.

C-64 can do in a flash.

Figure 6-10 is the listing of a subroutine for generating a full-screen menu such as the one shown in Fig. 6-8. Arguments of the subroutine are T$ (title), N (number of valid options, 1-N), and OPT$(1) through OPT$(N) (menu options).

To use this subroutine, the arguments are set as in any other subroutine, and then the subroutine is called with a GOSUB statement. Lines 10000-10070 in Fig. 6-10 do precisely this. When the subroutine is called, the screen is cleared, and the menu is generated. The subroutine PRINTs the title on the top row, computes the average length of an option and centers the option block horizontally, centers it vertically based on the number of options, and then PRINTs the prompt on the bottom row. The aesthetics of a menu with the title at the top of the screen and the prompt at the bottom are dubious at best, and most menus would look much nicer with the three parts of the menu brought closer together. It is fairly easy to modify the subroutine to change its layout. It is presented in the form shown in Fig. 6-10 mainly to provide a starting point in a menu-generation subroutine that you can later tailor to meet your own particular needs.

The subroutine is based on the code shown in Fig. 6-9, which was discussed earlier, and so I will focus here mainly on the differences.

Lines 4220-4230 generate the title.

Lines 4250-4290 compute the average length, L, of an option. Line 4250 initializes L to 0. Lines 4260-4280 then add up the total length of all options. Three is added to the length of each option for the number, period, and space that will be added later to precede the option. Line 4290 then divides the total by N (number of options) to compute the average length of an option. Line 4300 computes the horizontal tab position, H, required for PRINTing the options.

Line 4320 computes the first option's vertical offset (row number) based on the number of options. Line 4340 positions the cursor vertically, and lines 4350-4370 PRINT the prompts, adding a selection number, period, and space at the beginning of each. (One is subtracted from TAB position H to correct for the extra space BASIC adds to the front of a number when it converts to string form.)

Lines 4390-4480 PRINT the prompt and collect user keystrokes. These lines are identical to those in Fig. 6-9 except for line numbers and the use of the variable N in the error test at line 4450.

This subroutine permits you to create a full-screen menu in about as much time as it takes to type in lines with the subroutine's arguments. In this particular example, eight lines of code supply the arguments and subroutine call necessary to generate the five-option menu shown in Fig. 6-8. Using the subroutine is a considerable timesaver, and also eliminates the drudgery of counting the characters of the options and computing the horizontal tab positions and vertical offset.

Control Switching. Creating a menu directly or with a subroutine is an act of screen generation and does not control anything. In order to execute control, you must convert the user's option selection into a control action. That is, if the user selects option 1 by typing the 1 key and then pressing Return, you must send program control to the part of the program that executes the appropriate function. In this section I show two ways to switch control using BASIC statements: (1) IF-THEN logic, and (2) ON X GOTO statement. I also show how and why the two methods are sometimes combined. The code examples that follow use the Menu-Generation subroutine described in the previous section; however, these control-switching methods apply equally when a menu is generated directly.

Line 4430 of the Menu-Generation subroutine (Fig. 6-10) GETs the user's menu selection and assigns it to the string variable A$. Line 10180 evaluates A$ and assigns the result to the real variable A. Thus, the subroutine returns its result in two forms, and either can be used for control switching. Also note that since the subroutine does the error-testing—only valid values of A$ or A can be returned—further error-testing of these values is unnecessary.

Suppose that your program has the five options shown in Fig. 6-7 and that the lines at which these program functions start are as follows:

1. ENTER DATA — 12000

2. CREATE NEW FILE — 14000
3. PRINT REPORTS — 20000
4. EDIT DATA BASE — 24000
5. QUIT PROGRAM — 10500

You can use IF-THEN logic to control switching by inserting the following lines in the listing shown in Fig. 6-10:

```
10080 REM—PROGRAM CONTROL—
10090 IF A=1 GOTO 12000
10100 IF A=2 GOTO 14000
10110 IF A=3 GOTO 20000
10120 IF A=4 GOTO 24000
10130 IF A=5 GOTO 10500
```

The code just listed uses the real variable A to control switching. You can also use the string form of this variable:

```
10080 REM—PROGRAM CONTROL—
10090 IF A$="1" GOTO 12000
10100 IF A$="2" GOTO 14000
10110 IF A$="3" GOTO 20000
10120 IF A$="4" GOTO 24000
10130 IF A$="5" GOTO 10500
```

Not only can you use either the real or string form of the switching variable, but you could even combine them, if desired:

```
10090 IF A=1 GOTO 12000
10100 IF A=2 GOTO 14000
10110 IF A$="3" GOTO 20000
10120 IF A$="4" GOTO 24000
10130 IF A$="5" GOTO 10500
```

There may not seem to be much point in doing this, but it comes in handy, as I will demonstrate later.

Any one of these three sets of code will do the job for us, but the most economical is the first, which works with real numbers. Still, this requires five statements, one for each menu option.

The ON X GOTO statement is considerably more economical than IF-THEN logic. This statement switches control to the Xth statement following the GOTO. To illustrate, the switching logic just given can be reduced to this statement:

```
10010 ON X GOTO 12000, 14000, 20000, 240000,
24000, 10500
```

This is obviously much shorter, but it requires that X be a number. You cannot use a string in place of X. Further, if X is zero or a number greater than the number of arguments following GOTO, control falls through to the next line. For example, when this program is executed, control drops to line 30:

```
10 X=0
20 ON X GOTO 40, 50, 60
30 PRINT"LINE 30":END
40 PRINT"LINE 40":END
50 PRINT"LINE 50":END
60 PRINT"LINE 60":END
```

The same result is obtained if X is set equal to a number greater than 3.

Mixing Number and Letter Identifiers. It is sometimes useful to mix number and letter identifiers in menus. Suppose, for example, that you want to use a menu that has these four options:

1 — ENTER DATA
2 — CREATE NEW FILE
F1 — CHANGE MODE
F2 — BACK UP

In order to generate this menu, we must modify the Menu-Generation subroutine so that it allows us to define not only the option label (OPT$(A)), but also the option identifier, so that we can precede a number by anything. In addition, we must modify the subroutine's error test so that it accepts an A$ value of 0—this is the VALue of A$ if a nonnumber key is pressed. These changes are fairly easy to make.

Lines 4200-4490 of Fig. 6-11 contain a modified version of the Menu-Generation subroutine that provides the additional features needed. Changes were made to the following lines:

4270—Length computation based on OPT$(A) only, since it will be completely defined as an argument (formerly 3 was

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
1500 REM--CENTER & PRINT T$--
1510 T=(39-LEN(T$))/2
1520 PRINT CHR$(18);
1530 PRINT TAB(T)T$
1540 RETURN
4200 REM--FULL-SCREEN MENU GENERATOR--
4210 PRINT CHR$(147);
4220 T$=T$+" MENU"
4230 GOSUB 1500:REM PRINT TITLE
4240 REM-COMPUTE MENU STARTING COLUMN-
4250 L=0
4260 FOR A=1 TO N
4270 L=L+LEN(OPT$(A))
4280 NEXT
4290 L=L/N
4300 H=(39-L)/2:REM COMPUTE TAB
4310 REM-COMPUTE VERTICAL OFFSET-
4320 V=INT(25-N)/2
4330 REM-DISPLAY OPTIONS-
4340 SYS C0,0,V:REM POSITION FIRST OPTION
4350 FOR A=1 TO N
4360 PRINT TAB(H)OPT$(A)
4370 NEXT
4380 REM-PRINT PROMPT-
4390 SYS C0,1,24,17:REM CLEAR PROMPT LINE
4400 SYS C0,0,24,17:REM POSITION PROMPT
4410 PRINT CHR$(18)"CHOICE?";
4420 POKE 204,0:REM FLASH CURSOR
4430 GET A$:IF A$="" GOTO 4430:REM GET MENU SELECTION
4435 POKE 204,1:REM TURN FLASH OFF
4440 A=VAL(A$)
4445 IF A=0 GOTO 4490:REM NON-NUMBER KEY PRESSED
4450 IF A<1 OR A>N0 GOTO 4390
4460 PRINT CHR$(146)" ";A$;:REM DISPLAY KEYSTROKE
4470 GET B$:IFB$="" GOTO 4470:REM GET ENTRY VERIFICATION
4480 IF ASC(B$)<>13 GOTO 4390:REM VERIFICATION RETURN KEY?
4490 RETURN
10000 REM--DEFINE MENU--
10010 N=4
10020 N0=2
10030 T$="MASTER"
10040 OPT$(1)="1  - ENTER DATA"
10050 OPT$(2)="2  - CREATE NEW FILE"
10060 OPT$(3)="F1 - CHANGE MODE"
10070 OPT$(4)="F2 - BACK UP"
10080 GOSUB 4200
10090 IF A=1 GOTO 12000:REM KEY=1
10100 IF A=2 GOTO 14000:REM KEY=2
10110 IF A$=CHR$(133) GOTO 16000:REM KEY=F1
10120 IF A$=CHR$(137) GOTO 20000:REM KEY=F2
10130 GOTO 10000:REM KEY=INVALID--RE-GENERATE MENU
```

Fig. 6-11. A subroutine (lines 4200-4490) for generating a full-screen menu with any option indicators (alternative to Fig. 6-10).

added to account for the number, period, and space preceding the option label).

4360—TAB (H-1) changed to TAB (H) since no number to string conversion occurs.

4445—This line was added to permit a RETURN from the subroutine if the value of A is 0. A is 0 if a 0 is typed in, or if any nonnumber key is pressed. Since some such values of A correspond to invalid A$ entries, further error-testing must occur outside of the subroutine. Note that these entries are not verified by the Return key.

4450—The limiting argument N has been changed to N0 to allow for a greater number of options than valid numbers. For example, in this menu, there are four menu options, but only two valid option numbers (1, 2).

Lines 10000-10120 show the code for generating the actual menu. Line 10010 defines the number of menu options (N), and line 10020 defines the number of options with number identifiers (N0). Line 10030 defines the menu title. Lines 10040-10070 define the four menu options, including identifiers. Line 10080 calls the subroutine.

IF-THEN logic is used to switch control. Lines 10090-10120 look for keys 1, 2, F1, and F2, respectively. If no such key has been pressed, then line 10130 sends control back to line 10000, which regenerates the menu.

Menu Control Networks. Network is the technical term for the structure that results when you create a series of menus and then link them together. Figure 6-12 is an example of a simple three-level, hierarchical network. This particular type of network is referred to as a tree. The reason is that it branches out, downward from the top. Each menu, or node, is linked to only one menu above it, and to those below. The further down the tree you are, the more menus separate you from the top.

The tree structure is widely used in computer programs. It provides access to any part of the program, and its simplicity makes it easy to learn. It has one disadvantage, however. In a large network,

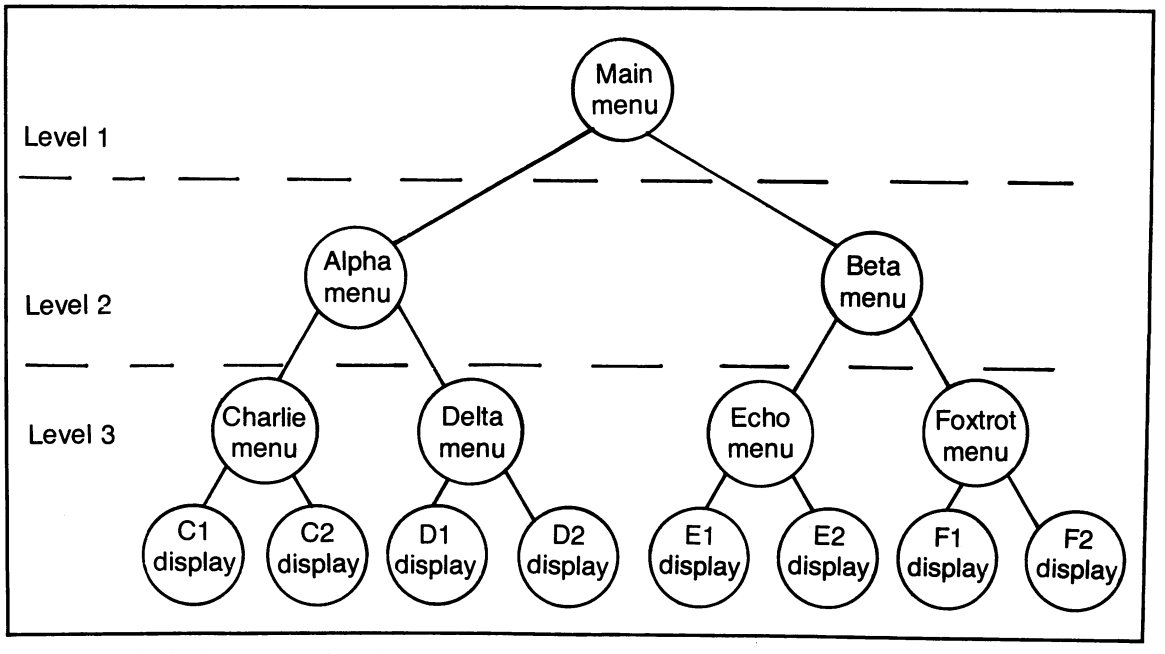


Fig. 6-12. A simple, hierarchical, three-level menu network in the form of a tree.

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
1500 REM--CENTER & PRINT T$--
1510 T=(39-LEN(T$))/2
1520 PRINT CHR$(18);
1530 PRINT TAB(T)T$
1540 RETURN
4010 REM--TEMPORARY PAUSE--
4020 SYS C0,0,24,6
4030 PRINT CHR$(18);
4040 PRINT "[PRESS SPACE BAR TO CONTINUE]";
4050 GET A$
4060 IF A$<>" " GOTO 4050
4070 PRINT
4080 RETURN
4200 REM--FULL-SCREEN MENU GENERATOR--
4210 PRINT CHR$(147);
4220 T$=T$+" MENU"
4230 GOSUB 1500:REM PRINT TITLE
4240 REM-COMPUTE MENU STARTING COLUMN-
4250 L=0
4260 FOR A=1 TO N
4270 L=L+LEN(OPT$(A))
4280 NEXT
4290 L=L/N
4300 H=(39-L)/2:REM COMPUTE TAB
4310 REM-COMPUTE VERTICAL OFFSET-
4320 V=INT(25-N)/2
4330 REM-DISPLAY OPTIONS-
4340 SYS C0,0,V:REM POSITION FIRST OPTION
4350 FOR A=1 TO N
4360 PRINT TAB(H)OPT$(A)
4370 NEXT
4380 REM-PRINT PROMPT-
4390 SYS C0,1,24,17:REM CLEAR PROMPT LINE
4400 SYS C0,0,24,17:REM POSITION PROMPT
4410 PRINT CHR$(18)"CHOICE?";
4420 POKE 204,0:REM FLASH CURSOR
4430 GET A$:IF A$="" GOTO 4430:REM GET MENU SELECTION
4435 POKE 204,1:REM TURN FLASH OFF
4440 A=VAL(A$)
4445 IF A=0 GOTO 4490:REM NON-NUMBER KEY PRESSED
4450 IF A<1 OR A>N0 GOTO 4390
4460 PRINT CHR$(146)" ";A$;:REM DISPLAY KEYSTROKE
4470 GET B$:IFB$="" GOTO 4470:REM GET ENTRY VERIFICATION
4480 IF ASC(B$)<>13 GOTO 4390:REM VERIFICATION RETURN KEY?
4490 RETURN
10000 REM--MAIN MENU--
10010 N=3
10020 N0=2
10030 T$="MAIN"
10040 OPT$(1)="1 - ALPHA"
10050 OPT$(2)="2 - BETA"
10060 OPT$(3)="F1- QUIT PROGRAM"
```

```
10070 GOSUB 4200
10080 IF A=1 GOTO 12000
10090 IF A=2 GOTO 19000
10100 IF A$=CHR$(133) THEN PRINT CHR$(147):END
10110 GOTO 10010
12000 REM--ALPHA MENU--
12010 N=3
12020 N0=2
12030 T$="ALPHA"
12040 OPT$(1)="1 - CHARLIE"
12050 OPT$(2)="2 - DELTA"
12060 OPT$(3)="F1- BACK UP"
12070 GOSUB 4200
12080 IF A=1 GOTO 13000
12090 IF A=2 GOTO 16000
12100 IF A$=CHR$(133) GOTO 10000
12110 GOTO 12010
13000 REM--CHARLIE MENU--
13010 N=3
13020 N0=2
13030 T$="CHARLIE"
13040 OPT$(1)="1 - C1 DISPLAY"
13050 OPT$(2)="2 - C2 DISPLAY"
13060 OPT$(3)="F1- BACK UP"
13070 GOSUB 4200
13080 IF A=1 THEN GOSUB 14000:GOTO 13000
13090 IF A=2 THEN GOSUB 15000:GOTO 13000
13100 IF A$=CHR$(133) GOTO 12000
13110 GOTO 13010
14000 REM--C1 DISPLAY--
14010 PRINT CHR$(147);
14020 T$="C1 DISPLAY"
14030 GOSUB 1500
14040 GOSUB 4010
14050 RETURN
15000 REM--C2 DISPLAY--
15010 PRINT CHR$(147);
15020 T$="C2 DISPLAY"
15030 GOSUB 1500
15040 GOSUB 4010
15050 RETURN
16000 REM--DELTA MENU--
16010 N=3
16020 N0=2
16030 T$="DELTA"
16040 OPT$(1)="1 - D1 DISPLAY"
16050 OPT$(2)="2 - D2 DISPLAY"
16060 OPT$(3)="F1- BACK UP"
16070 GOSUB 4200
16080 IF A=1 THEN GOSUB 17000:GOTO 16000
16090 IF A=2 THEN GOSUB 18000:GOTO 16000
16100 IF A$=CHR$(133) GOTO 12000
16110 GOTO 16010
17000 REM--D1 DISPLAY--
17010 PRINT CHR$(147);
17020 T$="D1 DISPLAY"
17030 GOSUB 1500
17040 GOSUB 4010
17050 RETURN
```

```
18000 REM--D2 DISPLAY--
18010 PRINT CHR$(147);
18020 T$="D2 DISPLAY"
18030 GOSUB 1500
18040 GOSUB 4010
18050 RETURN
19000 REM--BETA MENU--
19010 N=3
19020 N0=2
19030 T$="BETA"
19040 OPT$(1)="1 - ECHO"
19050 OPT$(2)="2 - FOXTROT"
19060 OPT$(3)="F1- BACK UP"
19070 GOSUB 4200
19080 IF A=1 GOTO 20000
19090 IF A=2 GOTO 23000
19100 IF A$=CHR$(133) GOTO 10000
19110 GOTO 19010
20000 REM--ECHO MENU--
20010 N=3
20020 N0=2
20030 T$="ECHO"
20040 OPT$(1)="1 - E1 DISPLAY"
20050 OPT$(2)="2 - E2 DISPLAY"
20060 OPT$(3)="F1- BACK UP"
20070 GOSUB 4200
20080 IF A=1 THEN GOSUB 21000:GOTO 20000
20090 IF A=2 THEN GOSUB 22000:GOTO 20000
20100 IF A$=CHR$(133) GOTO 19000
20110 GOTO 20010
21000 REM--E1 DISPLAY--
21010 PRINT CHR$(147);
21020 T$="E1 DISPLAY"
21030 GOSUB 1500
21040 GOSUB 4010
21050 RETURN
22000 REM--E2 DISPLAY--
22010 PRINT CHR$(147);
22020 T$="E2 DISPLAY"
22030 GOSUB 1500
22040 GOSUB 4010
22050 RETURN
23000 REM--FOXTROT MENU--
23010 N=3
23020 N0=2
23030 T$="FOXTROT"
23040 OPT$(1)="1 - F1 DISPLAY"
23050 OPT$(2)="2 - F2 DISPLAY"
23060 OPT$(3)="F1- BACK UP"
23070 GOSUB 4200
23080 IF A=1 THEN GOSUB 24000:GOTO 23000
23090 IF A=2 THEN GOSUB 25000:GOTO 23000
23100 IF A$=CHR$(133) GOTO 19000
23110 GOTO 23010
24000 REM--F1 DISPLAY--
24010 PRINT CHR$(147);
24020 T$="F1 DISPLAY"
24030 GOSUB 1500
24040 GOSUB 4010
```

```
24050 RETURN
25000 REM--F2 DISPLAY--
25010 PRINT CHR$(147);
25020 T$="F2 DISPLAY"
25030 GOSUB 1500
25040 GOSUB 4010
25050 RETURN
```

Fig. 6-13. Program code required to generate menus and dummy displays of control network shown in Fig. 6-12.

with many levels, it may take the user a long time to go between programs—particularly if the programs are in different branches of the network. For example, in Fig. 6-12, to go from Delta menu to Foxtrot menu, the user must go through three other menus first—Alpha menu, Main menu, and Beta menu.

By using the Menu-Generation Subroutine (Fig. 6-11), it is fairly easy to write the code that generates a network of menus such as shown in Fig. 6-12. There are seven menus in this network, but they and their links to other menus can be created in about as much time as it takes to write down their arguments, the subroutine calls, and the control logic—perhaps an hour or two for an average programmer.

To illustrate, Fig. 6-13 contains the code required to generate all of the menus as well as eight dummy displays. You may find it instructive to type this in and try it out for yourself—it shows how quickly you can put together a network of complete and professional-looking menus.

The Main Menu is generated on lines 10000-10110. Main Menu links to Alpha Menu (lines 12000-12120) and Beta Menu (lines 19000-19110). Alpha Menu links to Charlie Menu (lines 13000-13110) and Delta Menu (lines 16000-16110). Similarly, Beta Menu links to Echo Menu (lines 20000-20110) and Foxtrot Menu (lines 23000-23110).

Each of the four third-level menus—Charlie, Delta, Echo, Foxtrot—is further linked to two dummy displays.

The control logic used with these menus is of the IF-THEN type, and it permits the user to move downward by pressing a number key, or to BACK UP by pressing the F1 key.

Because of the time required to move between branches in a network such as Fig. 6-12, programmers often modify it by providing a quick return to the Main Menu. The resulting network then appears as shown in Fig. 6-14. A Main Menu option is added to all program menus (except the Main Menu itself).

You can also, of course, add additional options to specific menus to allow users to move between different parts of the program. A word of caution, however. A menu network is much like the logic of your program. If your program is well designed, and the programs relate to one another in a logical fashion, then it should not be necessary to do very much patching between different parts of the program. Each link between a network is analogous to a GOTO statement in a program. When you keep these GOTOs working according to a simple pattern, your program organization is easy to follow. As the links between menus become more arbitrary and disordered, your control network is analogous to program code that is written according to the nonrules of spaghetti logic. Just as you must "beware the GOTO" in writing program code, you must also beware linking menus in a disordered fashion.

The tree is not the only way to construct a menu network. You can, for example, construct one in the form of a ring or matrix (Fig. 6-15), or in other ways. In a ring, there is usually only one menu, and every option is displayed on it. Such a menu can get very long. If, however, all of the options are independent—that is, unrelated to each other in any hierarchial way—then a ring network may make sense.

A matrix, or net, allows essentially arbitrary connections among menus. This kind of network

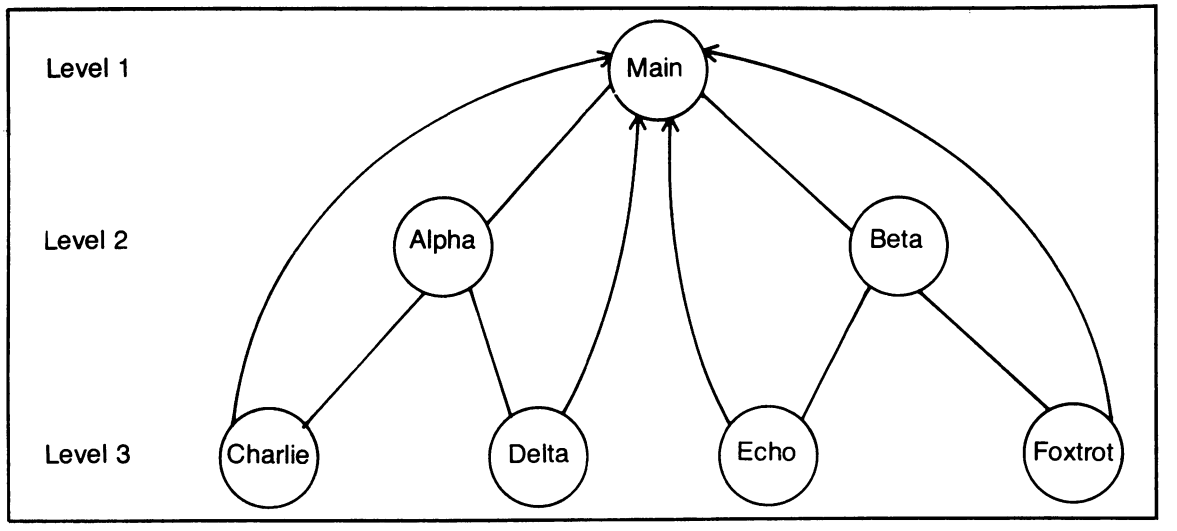


Fig. 6-14. A three-level hierarchical menu network in which all of the menus, even those at the bottom, have a direct route back to the Main Menu.

may make sense in some programs, but it would be extremely difficult to learn and use.

Actually, there is no ideal control structure that will work with all programs because the control structure should be derived from the interrelationships among the various functions in the program. Often these relationships are hierarchical. If so, it is natural to use a tree as the form of the control structure. The control structure should be derived from the program, not imposed upon it. To derive the structure, start with the subprograms, displays, functions, and other things your program does. Cluster, or group, related things together. Look for hierarchical relationships among things. Then plan the network.

Here are a few general guidelines for planning such networks:

- Use a hierarchical network, if possible.
- Try not to allow more than nine options per menu.
- If your network has three or more levels, provide an option on all menus that goes back to the top level (Main Menu or its equivalent).
- In general, it is better to design a network with fewer levels and more options per menu than one with a few options per menu and more levels. For example, if your program has a total of 64 functions, you are better off with two menu levels of eight options each (Fig. 6-16) than with three levels of four each (Fig. 6-17) or six levels of two each (Fig. 6-18).

Partial-Screen Menus

A partial-screen menu usually takes up only a few lines. Such a menu has a minimum of one line, and may have several. Figures 6-19 through 6-21 show three typical partial-screen menus. As the name indicates, partial-screen menus use only part of the screen—they share it with the information being displayed. The menu may be used to manipulate the display, to control data entry, or to perform other functions relating to the display. In this respect, it differs from a full-screen menu, which usually makes a clean break between display screens. While it is on the screen, other information is not.

Another way in which full-screen and partial-screen menus differ is in their use. Partial-screen menus are generally used locally, on a particular screen. They allow the user to exercise control of that screen. The full-screen menu is generally used globally, for moving around the program.

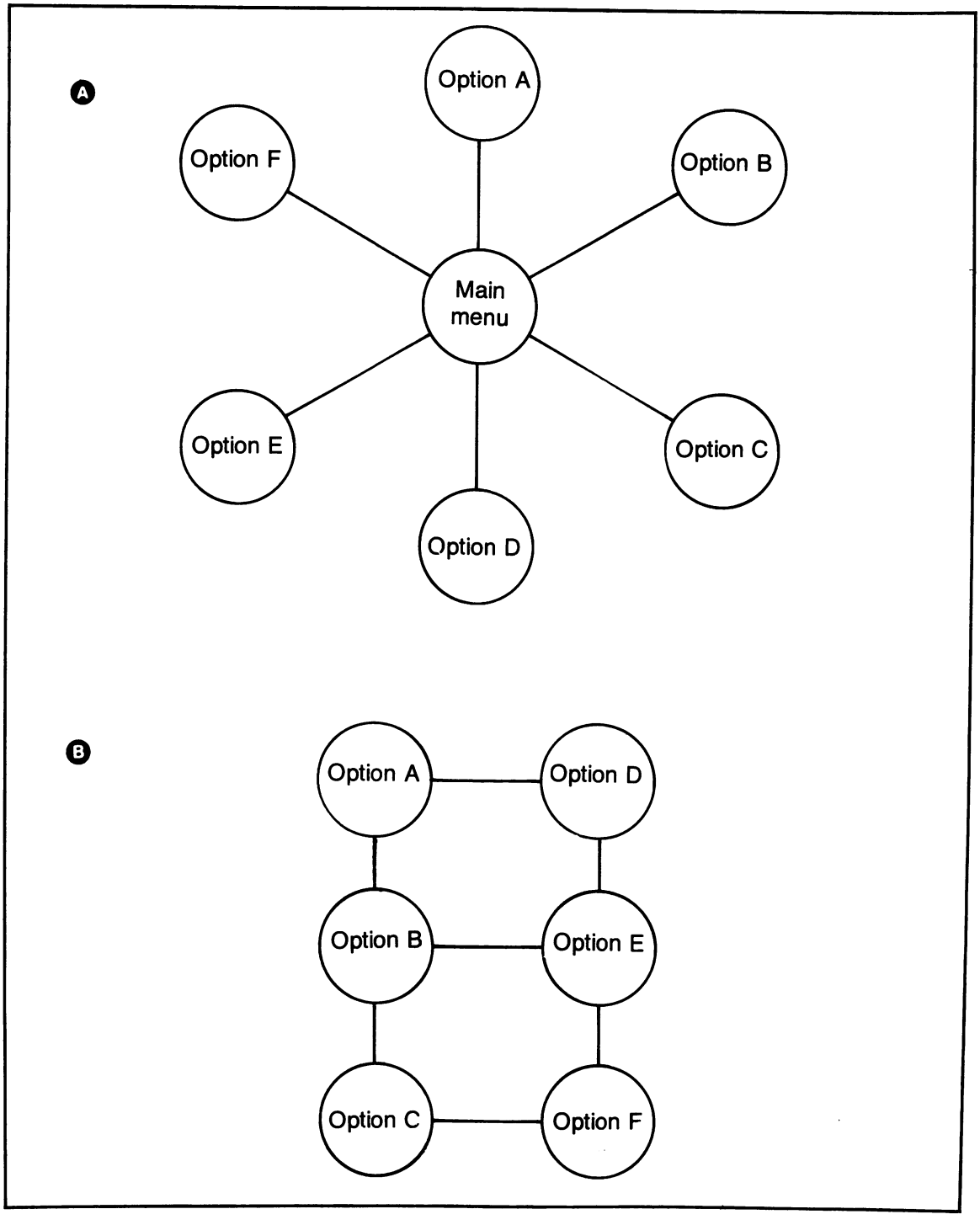


Fig. 6-15. A program control network in the form of (a) ring and (b) matrix.

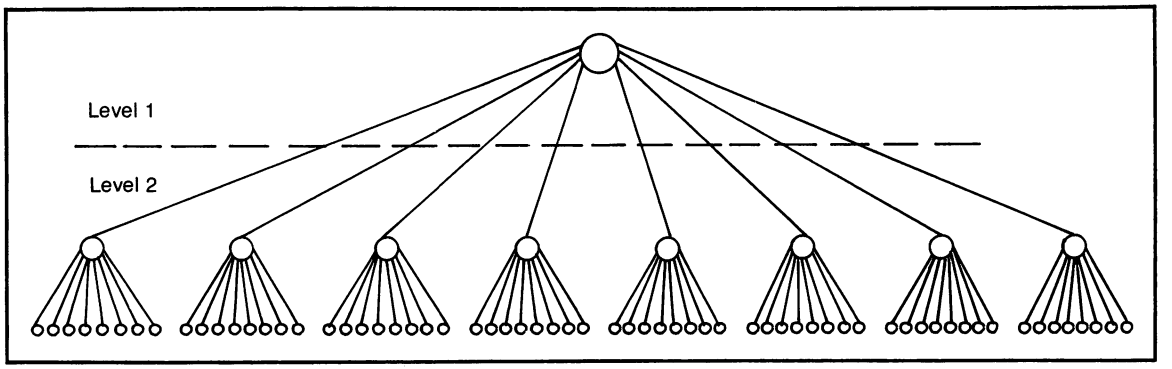


Fig. 6-16. A program control network in which the options are selectable from menus with eight options (breadth=8), and there are two levels of menus (depth=2).

The menu used in Fig. 6-19 to control what is on the display, obtain help information, page the display, or exit the display. The characteristics of this menu are:

- It takes up two lines at the bottom of the screen.
- It is in reverse video.
- Menu options are selected with letters which bear a mnemonic relationship to the options.
- Entries are single keystrokes, collected with the GET statement.
- The menu is untitled.
- There is no prompt.

A title and prompt could be included with the menu quite easily. Whether or not they are depends mainly upon space availability. Since the menu is not the focus of interest on the screen—the information on the screen is, and the menu plays second fiddle—it is often necessary to reduce it to the bare minimum. This may mean leaving out title and prompt, and reducing option labels to single words.

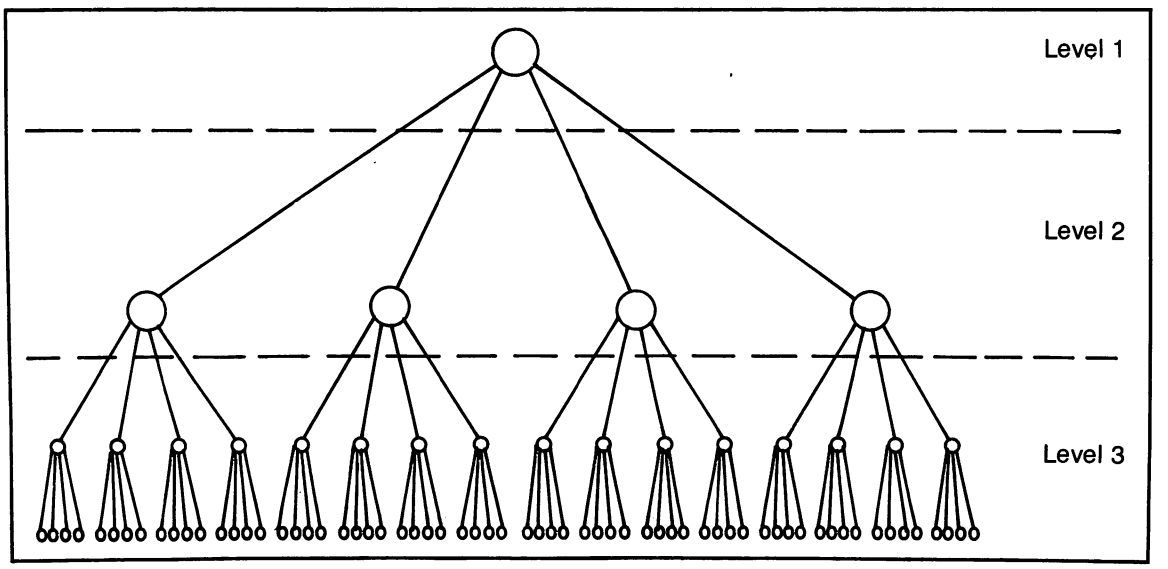


Fig. 6-17. A program control network in which the options are selectable from menus with four options (breadth=4), and there are three levels of menus (depth=3).

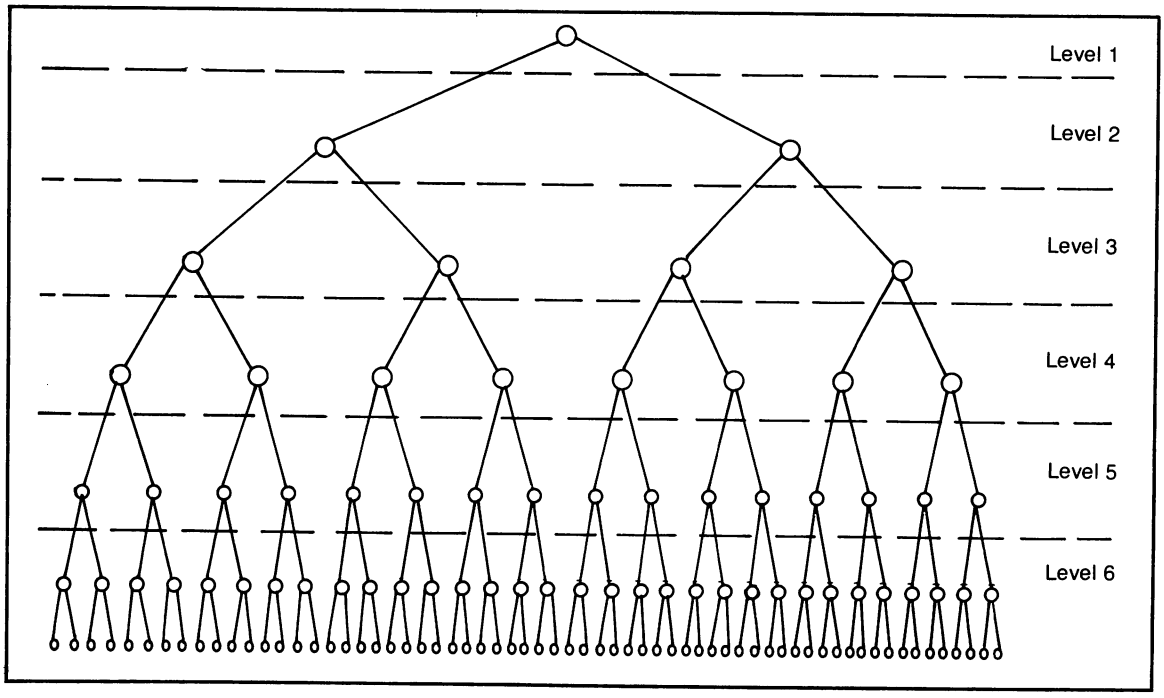


Fig. 6-18. A program control network in which the options are selectable from menus with two options (breadth=2), and there are six levels of menus (depth=6).

These menus can appear anywhere on the screen and take almost any form. Figure 6-20 shows a menu that is compressed horizontally instead of vertically, and which appears centered at the right edge of the display. This menu is used to control the graphics appearing in the center of the display.

To round out the picture, Fig. 6-21 shows a menu that appears at the top of the screen. This menu is used for the same purpose as that in Fig. 6-19, but is formatted and located somewhat differently. In general, the top of the screen is not the best place to locate a menu since this is where the title belongs. On some screens, however, it makes sense.

In all of these menus, the option identifier is a single letter. Numbers could be used, but letters are preferable on menus such as these. The identifiers should bear a mnemonic relationship to the option label. Menu format—a block in reverse video with a flashing cursor at the right edge—is designed to make the menu clearly recognizable as such without title or prompt. The menu could, of course, be created without reverse video or flashing cursor, but without these cues it would not be as distinct.

The identifiers in these menus are linked to the labels in three different ways:

- Letter in brackets (Fig. 6-19)—<A >ADD
- Equal sign (Fig. 6-20)—R=ROTATE
- Upper-lowercase (Fig. 6-21)—Quit

Any of these methods is effective, and mainly a matter of style. Once you have selected a method, however, stick with it consistently.

It is fairly easy to generate one of these menus either directly or with a subroutine. I will illustrate the technique with a subroutine, and you can adapt it to suit your particular needs. The subroutine allows you to generate a menu containing up to 24 lines, in reverse video, anywhere on the screen. Figure 6-22 contains a subroutine for generating a partial-screen menu. Arguments of this subroutine

are V (starting row), H (starting column), N (number of lines), and OPT$(1)-OPT$(N) (text of menu). The subroutine consists of lines 4600-4690.

Since the number of menu lines, N, is a variable, the subroutine must be able to create menus of different sizes. The subroutine uses a FOR-NEXT loop to control how many rows are cleared and lines printed. This work begins on line 4610 and ends on line 4660. Line 4620 clears the row; line 4630 positions the cursor at the beginning of the row, and line 4640 prints the Ath line of the menu in reverse video. Line 4650 then increments V by 1, so that the next row will be cleared and the line printed in the correct location when the FOR-NEXT loop recycles.

Line 4670 prints a flashing cursor, and line 4680 GETs the single character A$.

Note that this subroutine does no error-testing and does not echo the keystroke to the menu. It can easily be modified to do both of these things, however.

To illustrate the subroutine's application, lines 10000-10060 contain the argument assignments and subroutine call necessary to generate the menu shown in Fig. 6-19. Lines 10010-10050 assign the arguments. This is a two-line menu and lines 10040 and 10050 define the menu's two lines. Line 10060 calls the subroutine to generate the menu. The only tricky part is to make sure that every line of the menu (except the last) is of length 41-H. The last line must have one character less to allow room for the flashing cursor. If your menu starts at the left edge of the display (H=1), then upper menu lines must be 40 characters long and the last line 39 characters long. If the menu starts at column 21, then the upper lines must be 20 characters long and the last line 19 characters long. If you ignore these limits, your menu will be displayed, but formatted correctly.

SIMPLE CHOICES

After menus, the simple choice is a cinch.

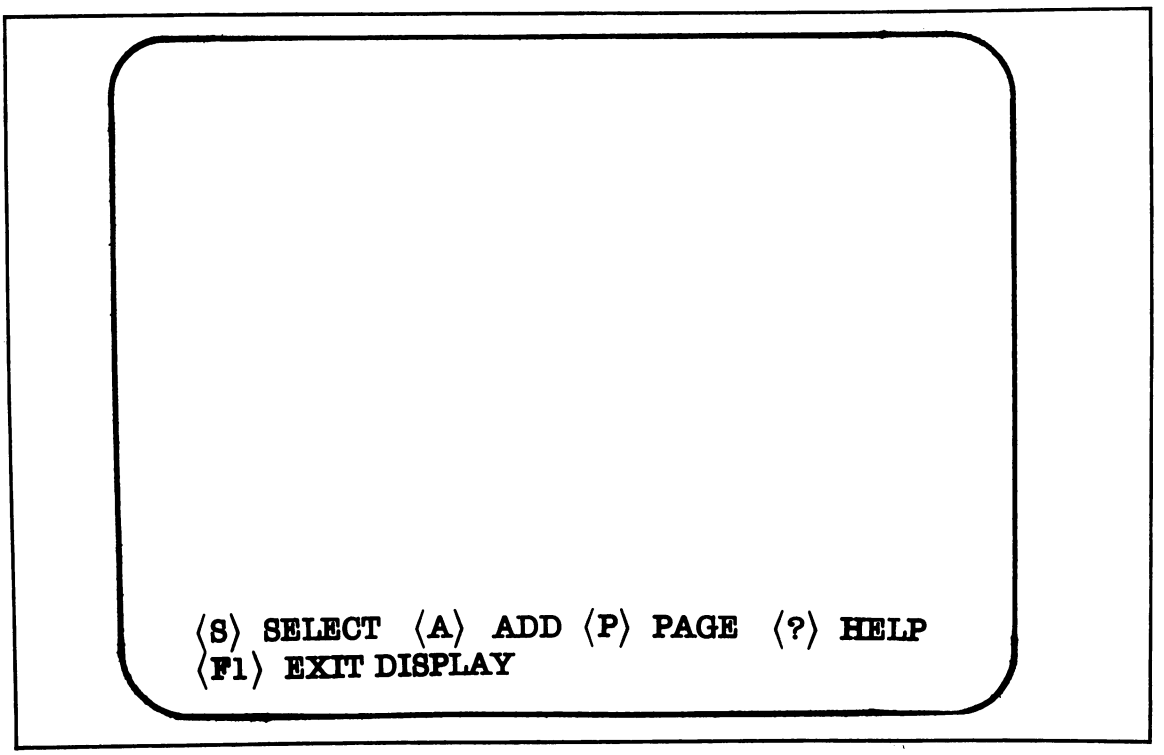


Fig. 6-19. Partial-screen menu consisting of two lines at the bottom of the display.

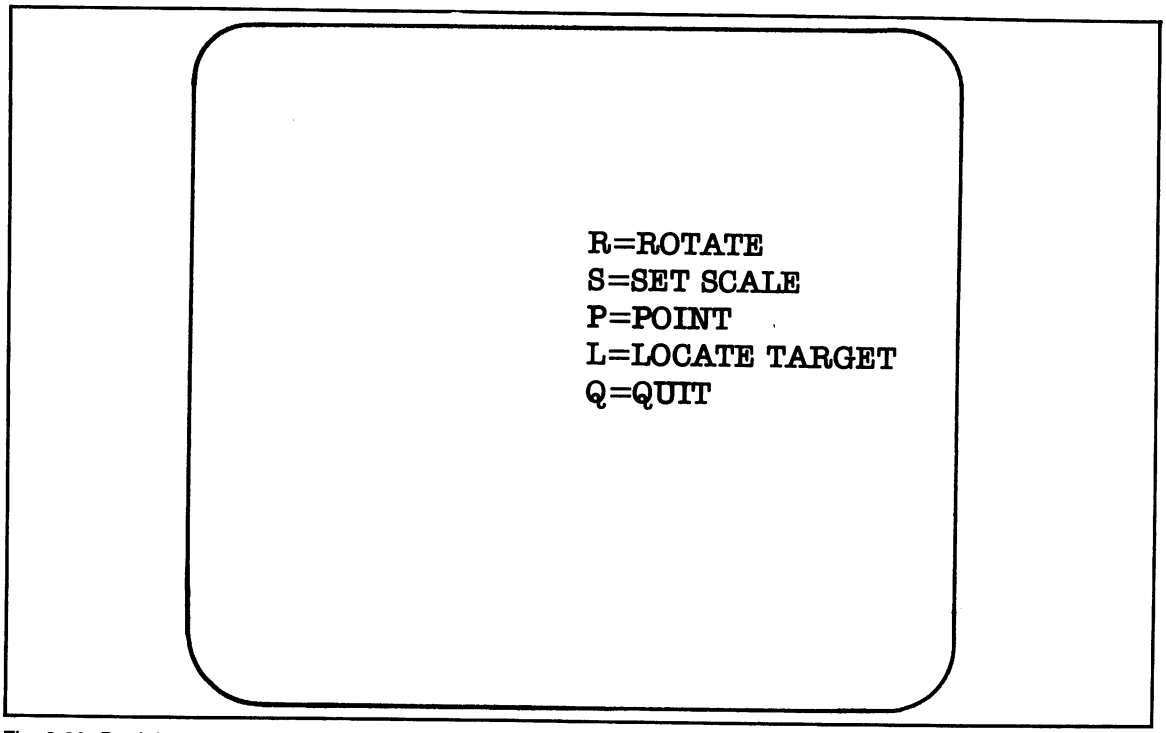


Fig. 6-20. Partial-screen menu consisting of a block of options on the right edge of display.

Here are some examples of control prompts that require the program user to make a simple choice:

- DO YOU WANT TO CONTINUE? (Y/N)
- DO YOU WANT DIRECTIONS? (Y/N)
- ARE YOU SURE? (Y/N)

In each of these examples, the user has come to some point in the program where a decision must be made. The easiest way to obtain the decision from the user is to display a short, two-choice question on the screen. The question may concern whether or not a hard-copy report should be generated, a file purged, or a particular program executed. A two-way question is much simpler to display than a two-way menu.

Of course, you could use a menu for getting the answer to such questions, but my feeling is that this is overkill. In general, it is best to reserve menus for situations that involve at least a three-way choice.

At the same time, you could use simple, two-way choices in place of menus, if you desired. Any program that can be written with menus can be rewritten so that it uses two-way choices instead. To illustrate, suppose that you start with the menu shown in Fig. 6-7. You can restructure it as the following series of two-way questions:

- DO YOU WANT TO UPDATE YOUR FILES? (Y/N)
- DO YOU WANT TO ANALYZE YOUR DATA BASE? (Y/N)
- DO YOU WANT TO EDIT YOUR DATA BASE? (Y/N)
- DO YOU WANT TO RECONCILE YOUR ACCOUNTS? (Y/N)
- DO YOU WANT TO QUIT THE PROGRAM? (Y/N)

The way this works is as follows. When the program starts, the first question is posed. If the user types in Y, then that program is executed.

Afterward, the first question is again posed.

If the user types in N to the first question, the second question is posed. If the user types in N, then the third question is posed, and so on, until the last question is asked. If the user types in N to that, then the series of questions begins over again at the first question. The questions continue to cycle in that manner, as long as the user responds N to each one.

If a program has more than a few options, this type of control is very cumbersome and slow. Can you imagine what it would be like if your program had a dozen different options and you were interested in doing number 11? If you happened to press the wrong key when the right question popped up, it would be kick-the-computer or strangle-the-programmer time.

If your program has fewer than, say, six options, then simple choices may be a good way to control it. Answering a question is straightforward and in many ways easier than dealing with a menu.

Even if your program does not use simple choices for overall control, it may use such choices for collecting answers to certain questions such as those posed earlier. In these situations, the simple choice is ideal.

The code for creating one of these prompts is straightforward, and I leave it to readers to write their own routines.

TYPED-IN CHOICES

Have you ever worked with a minicomputer or mainframe that required you to make use of a command language? You may have, for example, used the UNIX or other sophisticated operating system that provided no menus or other on-screen prompting, but that required you to type in special system commands that you had memorized. The UNIX operating system, and others like it, are very

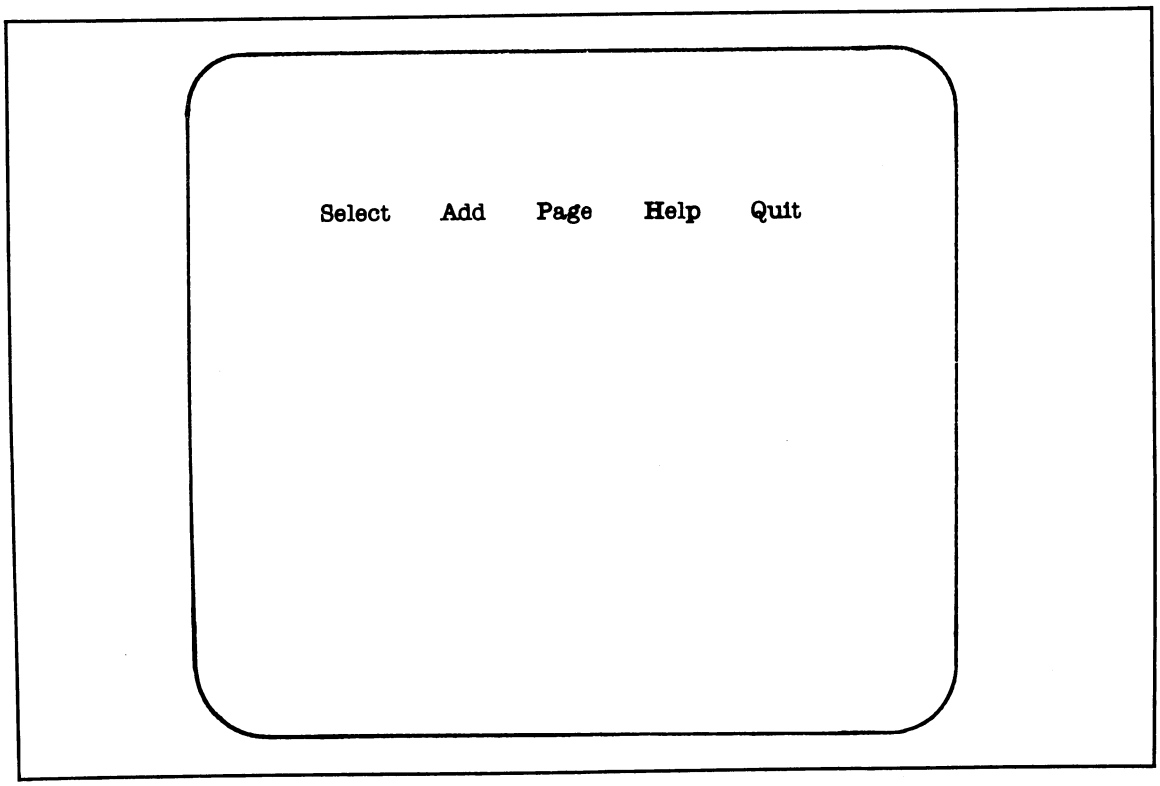


Fig. 6-21. Partial-screen menu consisting of one line at the top of the display.

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
4600 REM--PARTIAL-SCREEN MENU GENERATOR--
4610 FOR A=1 TO N
4620 SYS C0,1,V,H:REM CLEAR LINE
4630 SYS C0,0,V,H:REM POSITION CURSOR
4640 PRINT CHR$(18)OPT$(A);:REM PRINT MENU ROW A
4650 V=V+1
4660 NEXT
4670 POKE 204,0:REM FLASH CURSOR
4680 GET A$:IF A$="" GOTO 4680
4690 RETURN
10000 REM--2-LINE MENU--
10010 V=22
10020 H=5
10030 N=2
10040 OPT$(1)=" <A> ADD  <M> MODIF.  <D> DELETE    "
10050 OPT$(2)=" <Q> QUIT                           "
10060 GOSUB 4600
10070 END
```

Fig. 6-22. A subroutine (lines 4600-4690) for generating a partial-screen menu.

powerful. They let users tailor the tasks they want to perform very specifically. All the user must know is what statements to type in, and all the power of the computer is available.

The drawback is that users must memorize everything beforehand. There are no helpful menus to jog the memory, no safety nets to correct errors or prompt users along in the right direction. Users are very much on their own. There is a penalty to pay for such power. Still, computer experts much prefer command languages to those with extensive on-screen prompting.

Can you use a command language in your C-64?

Surely, it is possible, at least to some degree. To my knowledge, no one has yet written a version of the UNIX operating system for the C-64, nor do I think that it is very likely. Still, some of what such commond languages allow users to do is possible with the C-64.

To illustrate, let us go back, for a moment, to the three-level, hierarchical menu network illustrated in Fig. 6-12. Recall that the user starts off at the Main Menu, selects an option from it to go to a level 2 menu, and then selects an option from that to go to a level 3 menu. Once at a level 3 menu, the user can select one of the eight displays: C1, C2, D1, D2, E1, E2, F1, or F2. (Note that these displays could be replaced by subprograms or other program functions).

Now suppose that we simply eliminate the menus. Instead of using them, we will present a prompt that looks like this:

PROGRAM NAME:

To select a display, the user types in its name. We will write code to collect the keystrokes and assign them to a variable, and then use IF-THEN logic to switch control to the part of the program requested.

Figure 6-23 is a modified version of the program shown earlier in Fig. 6-13. The program in Fig. 6-13 uses menus to switch control, but the one in Fig. 6-23 uses typed-in choices.

The INPUT# data-entry subroutine beginning at line 3000 was used in Fig. 6-23 so that we have a

```
10 REM ************************************
20 REM * NOTE: THIS PROGRAM REQUIRES      *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
990 GOTO 10000
1500 REM--CENTER & PRINT T$--
1510 T=(39-LEN(T$))/2
1520 PRINT CHR$(18);
1530 PRINT TAB(T)T$
1540 RETURN
3010 REM--INPUT# DATA ENTRY--
3020 SYS C0,1,V,H :REM CLEAR LINE
3030 SYS C0,0,V,H:REM POSITION PROMPT
3040 PRINT CHR$(18);:REM REVERSE ON
3050 PRINT P$+":";:REM PRINT PROMPT
3060 PRINT CHR$(146);:REM REVERSE OFF
3070 OPEN 1,0
3080 INPUT#1,A$
3090 PRINT
3100 CLOSE 1
3110 RETURN
4010 REM--TEMPORARY PAUSE--
4020 SYS C0,0,24,6
4030 PRINT CHR$(18);
4040 PRINT "[PRESS SPACE BAR TO CONTINUE]";
4050 GET A$
4060 IF A$<>" " GOTO 4050
4070 PRINT
4080 RETURN
10000 REM--DATA ENTRY--
11000 REM--TYPED-IN DISPLAY SELECTOR--
11010 PRINT CHR$(147)
11020 V=12
11030 H=10
11040 P$="PROGRAM NAME"
11050 GOSUB 3010
11060 IF A$="C1" THEN GOSUB 14000
11070 IF A$="C2" THEN GOSUB 15000
11080 IF A$="D1" THEN GOSUB 17000
11090 IF A$="D2" THEN GOSUB 18000
11100 IF A$="E1" THEN GOSUB 21000
11110 IF A$="E2" THEN GOSUB 22000
11120 IF A$="F1" THEN GOSUB 24000
11130 IF A$="F2" THEN GOSUB 25000
11140 IF A$="QUIT" THEN PRINT CHR$(147):END
11150 GOTO 11000
14000 REM--C1 DISPLAY--
14010 PRINT CHR$(147);
14020 T$="C1 DISPLAY"
14030 GOSUB 1500
14040 GOSUB 4010
14050 RETURN
15000 REM--C2 DISPLAY--
15010 PRINT CHR$(147);
15020 T$="C2 DISPLAY"
15030 GOSUB 1500
15040 GOSUB 4010
```

```
15050 RETURN
17000 REM--D1 DISPLAY--
17010 PRINT CHR$(147);
17020 T$="D1 DISPLAY"
17030 GOSUB 1500
17040 GOSUB 4010
17050 RETURN
18000 REM--D2 DISPLAY--
18010 PRINT CHR$(147);
18020 T$="D2 DISPLAY"
18030 GOSUB 1500
18040 GOSUB 4010
18050 RETURN
21000 REM--E1 DISPLAY--
21010 PRINT CHR$(147);
21020 T$="E1 DISPLAY"
21030 GOSUB 1500
21040 GOSUB 4010
21050 RETURN
22000 REM--E2 DISPLAY--
22010 PRINT CHR$(147);
22020 T$="E2 DISPLAY"
22030 GOSUB 1500
22040 GOSUB 4010
22050 RETURN
24000 REM--F1 DISPLAY--
24010 PRINT CHR$(147);
24020 T$="F1 DISPLAY"
24030 GOSUB 1500
24040 GOSUB 4010
24050 RETURN
25000 REM--F2 DISPLAY--
25010 PRINT CHR$(147);
25020 T$="F2 DISPLAY"
25030 GOSUB 1500
25040 GOSUB 4010
25050 RETURN
```

Fig. 6-23. Program code for permitting user to select options directly, without using menus.

convenient way to generate the required prompt. The Menu-Generation subroutine (starting at line 4200) has been eliminated, along the menu generation code starting after line 10000. What remains of the old program are the dummy display-generating subroutines beginning at lines 14000, 15000, 17000, 18000, 21000, 22000, 24000, and 25000.

The new, controlling part of the program consists of lines 11000-11150. Lines 11020-11040 assign the arguments for generating the prompt that says "PROGRAM NAME" and printing it at row 12 and column 10 of the display. Line 11050 calls the INPUT# Data-Entry subroutine, which generates the prompt. After the subroutine call, lines 11060-11140 perform IF-THEN tests on the entry and switch control to the part of the program that contains the requested display. (The user types in "QUIT" to quit the program—see line 11140.) If all tests are failed, control falls through to line 11150, which sends it back to line 11000. This line leads to regeneration of the prompt and another data entry.

If the user types in a valid entry—say, C1—then one of the IF-THEN tests shifts control to the appropriate part of the program by calling one of the dummy display-generating subroutines. After the subroutine is executed, control RETURNs to

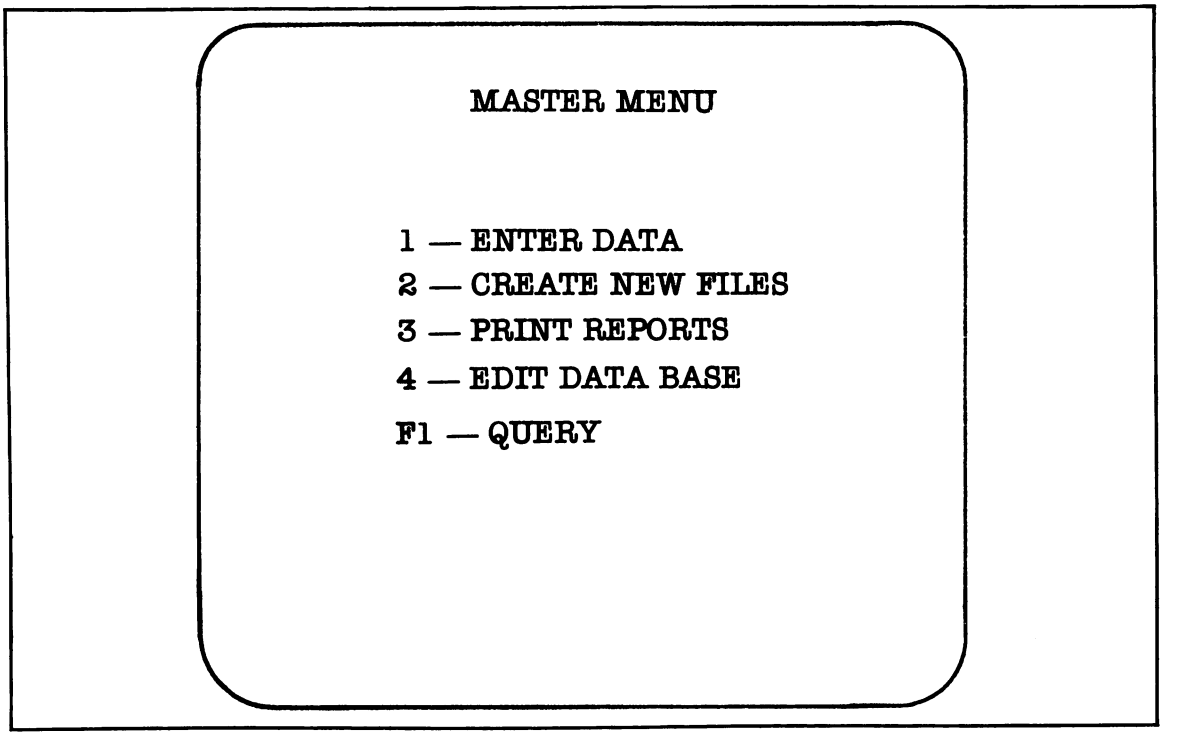


Fig. 6-24. Program control menu with F1 - QUERY option.

the line following the successful IF-THEN test, and eventually leads to a regeneration of the prompt.

Why would you want to control a program in this manner?

The main reason is to increase speed. Clearly, it is more difficult to remember names and type them in than it is to look up the name of a program on a menu and type in a single number. If your program gets big enough, and has enough options, however, working your way through the menus takes a lot of time. It may be much quicker to type in the name and be done with it.

Obviously, this does not work well in every program or for every program user. It is best for frequent users—those who learn the program very thoroughly and can get by without menus.

COMBINING MENUS AND TYPED-IN CHOICES

Would it ever make sense to combine menus and typed-in choices? Suppose, for example, that your program's Main Menu looks like the one shown in Fig. 6-24. This menu looks ordinary enough, but its last option is F1-QUERY. When a user types F1, the menu disappears and this prompt appears:

PROGRAM NAME:

Now the user can type in the program name and call it in the manner described in the previous section.

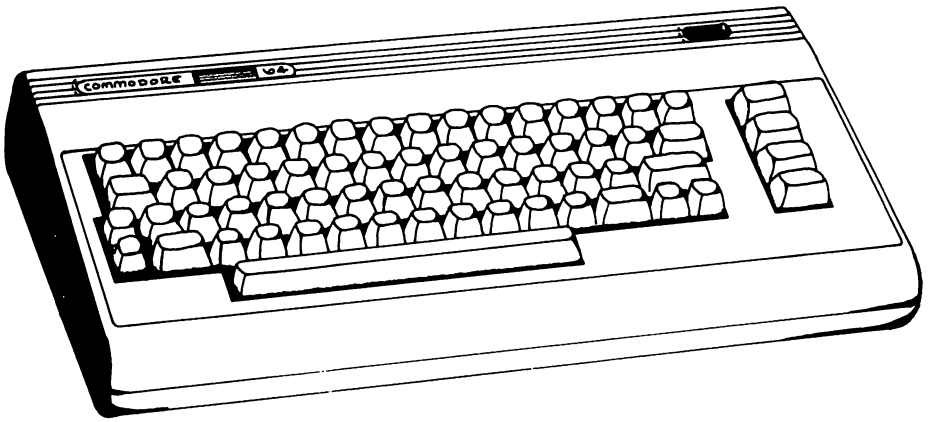
On the other hand, if the user wants to go back to using menus, typing in the word "MENU" puts the Main Menu on the display.

You get the idea. There are two ways to control the program, and the user can select either one. Why write a program that allows the user to do this? The advantage is that it lets the program be used at either a novice or expert level. The novice user can use menus. After using the program for a while, the novice becomes an expert. The menus can then be dispensed with. In short, the program grows with the user. This is a good quality for a program to have.

Still, not every program requires it. It begins to make sense, I believe, when your program is both complex, and users spend a lot of time working with it. If both of these conditions are satisfied, then users will probably benefit from being able to use menus or not, at their option.

Writing the code to permit dual control of a program is a simple extension of the techniques illustrated in this chapter. I leave it to your creativity, imagination, and expert judgment.

Chapter 7



Program Chaining

Program chaining is a technique used to link separate programs together. As programs grow in complexity, they also grow in size. Since your C-64 has a fixed amount of memory, there is a limit to how big a program can get before it uses up all available memory. In addition, the more variables you use, and the more files you load into memory, the less memory is left for your program. You may reach a point where your program fits, but there is not enough room for the data you want to process, or vice versa.

When you calculate everything beforehand and discover that both program and data will not fit into memory, you may decide to make the program smaller or to reduce the amount of data that it must process. In other words, you reduce the ambitiousness of your programming project. If you do not anticipate this, and the last byte of memory gets gobbled up while your program is running, then your ambitiousness is reduced for you, as the screen stares silently at you. If you are lucky, an out-of-memory message appears on your screen. If not, your computer slips into a state of catatonia and no longer responds to your ministerings.

There is a solution to such problems. Instead of writing one long program, break it up into smaller programs that are loaded into memory separately. This leaves memory to play with. This technique is widely used in business programs and is referred to as chaining. Most advanced BASICs have a CHAIN statement, or its equivalent. The C-64, unfortunately, does not. Nevertheless, there are techniques that allow you to chain. They are covered in this chapter.

When you break up one big program into two or more smaller ones, you are modularizing it on a global level. There are good and bad ways to do this modularization, and the good ones involve a bit more than slicing a listing in half with a pair of scissors. For this reason, this chapter begins with a discussion of some typical ways to modularize a program. The second section describes the program code required to chain; this is the mechanical part of getting from Program A to Program B unscathed. The final section introduces program verification.

If you want to chain to a program, it had better be on the disk that is in the drive because, if it is not, your first program will crash. Techniques for verifying that it is are covered in this section.

PROGRAM MODULARIZATION

I first discussed program modularization in Chapter 3. I stressed the importance of developing a program in a modular fashion, one part at a time, perfecting each part as you go. Later on, you link the various modules together to integrate them.

Some program developers prefer to break up their program into modules which are actually separate programs. One advantage of this is that, once you develop a module, it is easier to test and evaluate when it is separate from the rest of your program. You can make a copy of it on disk and give it to someone else to test more easily this way. In many ways, a program designed this way is easier to troubleshoot and maintain. When a problem occurs, it can usually be localized to the particular module that failed.

A drawback of this approach, however, is that it almost always takes up more disk space than a program developed as one continuous piece of code. Earlier in this book I remarked that many programs make use of common subroutines, and that these subroutines might comprise as much as 50 percent or more of any given program. If you break up your program into separate modules, then many of these subroutines must be duplicated in each module, thereby taking up more disk space than if the program were in one piece.

Another advantage of developing a program as one piece is speed. If your program is broken up into separate modules, and you must chain between them, then it is always slower. Each time your program chains from module A to module B, it must load in the new module. Doing this takes time. If, instead of chaining, you simply send control to a different part of one long program, only a fraction of a second is required. This is obviously much faster.

While there may be certain advantages for the developer in creating a program consisting of separate modules, the loss of speed is an important disadvantage for the user. It is hardly an even trade. All things considered, speed for the user is far more important than convenience for the program developer. Unfortunately, you do not always have a choice. If you want to include certain features in your program, and you are memory limited, you must modularize it and link the modules by chaining.

What is the best way to modularize a program? The answer depends upon the program, and you must do a little analysis to find the answer.

Start by noting the importance of speed. You must modularize the program in the way that allows the greatest speed. There is, however, a trade-off:

- The bigger you make a module, the more time it takes to load that module during chaining.
- The bigger you make a module, the more you can put into it, and the faster the execution of any given subprogram within that module. Alternatively, the shorter you make a module, the faster it loads but the less can go in it.

There are no hard and fast rules for how to break up a program into modules, but there are two very useful criteria to consider: (1) frequency of use of a subprogram, and (2) dependencies between subprograms.

The more frequently a particular subprogram is used, the more important it is to provide quick access to it. It follows that it makes sense to combine the most frequently used subprograms into a common module, if possible, so that the user does not have to chain to them separately.

If there are dependencies between subprograms—for example, between Data Entry and Data Base Edit—then combine them in a common module so that the user can move quickly between them without chaining.

To illustrate how you might apply these criteria to actual problems, assume that you are at the initial design stage on two different programs. Each of these programs needs five subprograms. You have done an analysis and determined the amount of memory required by each subprogram,

Subprogram	Memory (Kilobytes)	Frequency	Dependency
1	5	High	--
2	3	High	5
3	22	Low	--
4	24	Medium	--
5	9	Low	2

Table 7-1. Results of Analysis of Program 1.

the probable frequency of use, and the dependencies between subprograms. The results of the analysis are shown in Table 7-1 (Program 1) and Table 7-2 (Program 2).

Start with Program 1. One way to modularize this program is shown in Fig. 7-1. Subprograms 1, 2, and 5 are included in a common module because of the high frequency of use of Subprograms 1 and 2 and the dependency between Subprograms 2 and 5. Subprograms 3 and 4 are each included in separate modules because of the large amount of memory each requires. The main part of the program, then, consists of three large modules, a sort of triad. Each of these modules is linked to the other two, and can chain to either one. This implies that the main menu (or other controlling mechanism) is duplicated in all three modules.

There is also a fourth, Initialization module, which is linked solely to the module containing subprograms 1, 2, and 5. This module is optional, but is often either convenient or necessary in a program that is modularized. The program begins at this module, which dimensions arrays and does the other initialization preceding the working part of the program. By having it in a separate module, it can do its job and then get out of the way, without being carried as baggage in the working part of the program. Having it separate also helps avoid such inconveniences as redimensioned array errors.

Now consider Program 2. Table 7-2 shows that Program 2's memory, frequency, and dependency profile differs significantly from that of Program 1. The biggest difference is subprogram memory requirements. The subprograms are so

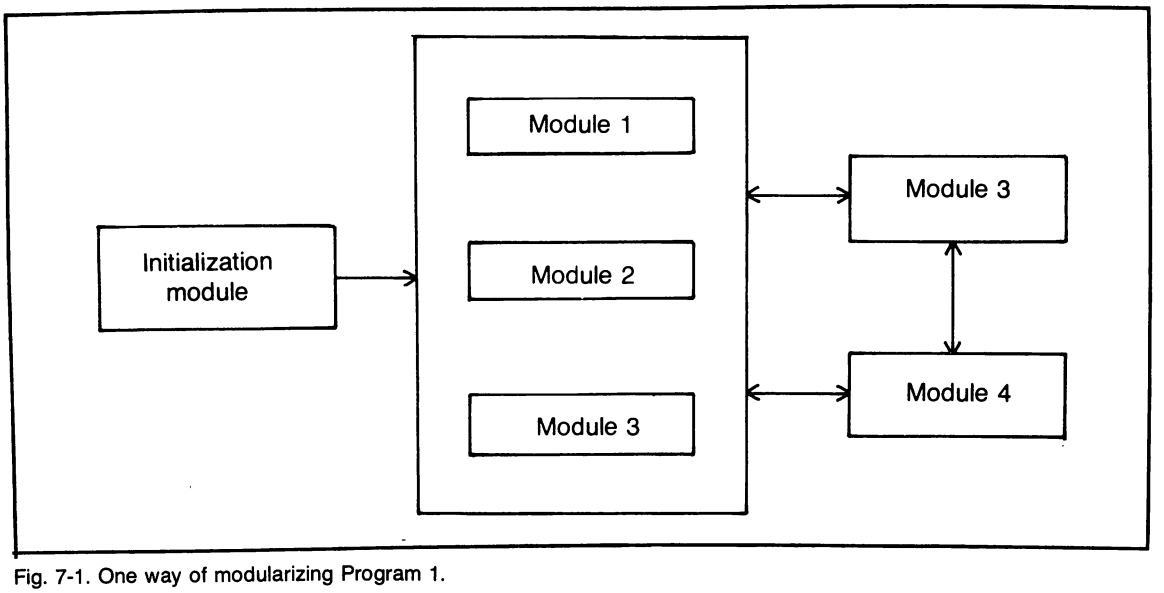


Fig. 7-1. One way of modularizing Program 1.

Subprogram	Memory (Kilobytes)	Frequency	Dependency
1	22	High	--
2	21	Low	--
3	25	Low	--
4	24	Medium	--
5	29	Low	--

Table 7-2. Results of Analysis of Program 2.

large that none can be combined. Each subprogram must itself become a separate module. The resulting design is shown in Fig. 7-2. The central core of this design is a Menu module. When the program starts, the Initialization module loads the Menu module. From there, the user can access any other module. When the user finishes with one of these modules, he or she returns again to the Menu module. The next subprogram is then executed, and so on.

These two examples are fairly typical of the way that subprograms are organized when chaining is used. Program 2 is, of course, the worst case—every subprogram gets its own module. Program 1 combines several subprograms into a few modules. Both of these designs use an Initialization module, which is optional, but generally a good thing to include. Another important difference between Program 1 and Program 2 is where the Main Menu is located. In Program 1, it is duplicated in every module, and every module is linked to every other module. In Program 2, the Main Menu is in a separate module of its own, and only that module is shared among all modules.

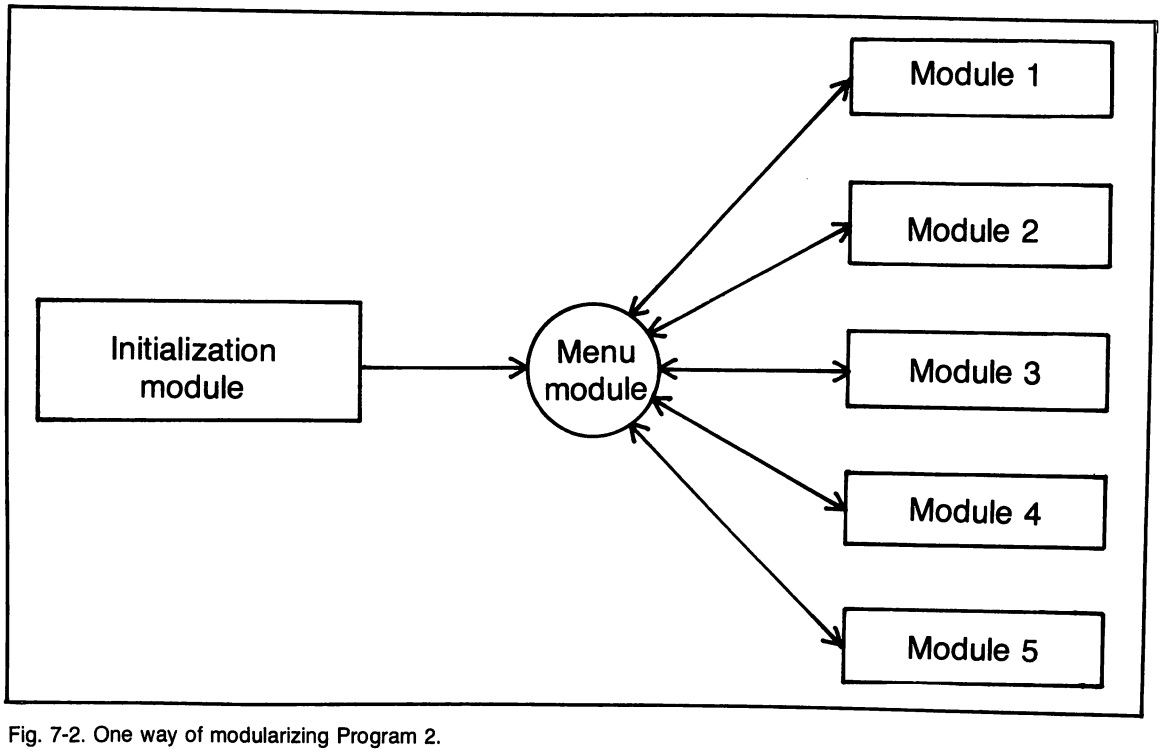


Fig. 7-2. One way of modularizing Program 2.

These examples, although simple, illustrate the types of analyses you must perform to modularize a program. Once you have performed the analyses and made the decision, you must link the modules together—that is, write the code required to chain.

HOW TO CHAIN

It is very easy to link two programs together if all you want to do is go from one program to the next and are not concerned with transferring variables or arrays. A little more is required to transfer variables and arrays, and still more is required to transfer strings. In short, there are three progressively more powerful ways to chain, depending upon your goal. This section discusses each of these, in turn.

Going From Program A to Program B

The C-64 Programmer's Reference Guide advises, on page 49, that you can use the LOAD statement to chain two programs together. The procedure described there is simple. Just include the following as the last line executed during the first program:

LOAD "PROGRAM B NAME",8

PROGRAM B NAME is, of course, the name of the second program. If you add this line, and then RUN Program A, you will see that Program B LOADs and begins executing. No problem, unless you want to transfer the values assigned to the variables, arrays, and strings used in the first program. Unless you are lucky, these values turn to garbage when the second program starts.

Actually, a little more than luck is involved. The deciding factor as far as variables and arrays are concerned is whether the second program is smaller than the first. If it is, then the variables and arrays transfer nicely, although the strings are usually lost. If you do not care about the strings, however, then there is no problem. In short, if the first subprogram in your program is the largest, and you do not care about transferring strings, then the LOAD statement, as described above, is the answer.

```
10 REM PROGRAM A
20 A=100
30 A(5)=555.55
40 A$="A STRING"
50 A$(5)="A STRING OF FIVE"
60 LOAD"PROGRAM B",8
```

Fig. 7-3. Program A.

To illustrate this simple way to chain, try it for yourself. Figure 7-3 contains a little program that assigns a real variable (A), one element of a real array (A(5)), a string (A$), and one element of a string array (A$(5)). You do not have to worry about dimensioning the arrays, since BASIC does it automatically. Type in the listing and SAVE it to disk as PROGRAM A.

Line 60 of Program A LOADs "PROGRAM B". The listing of Program B is shown in Fig. 7-4. This program attempts to PRINT the values of the four variables assigned by Program A. Type in Program B and SAVE it to disk as PROGRAM B.

Now LOAD and RUN Program A and see what happens.

Your C-64 should have suffered a serious case of amnesia, and may also have felt that one or more of the array variables had a bad subscript.

As this exercise illustrates, you can go from Program A to Program B by simply including a line at the end of the first program that LOADs the second. As this also shows, however, you will lose the real and array variables you were using, and may encounter some redimensioned array errors.

Transferring Variables and Arrays

If you want to transfer variables and arrays (nonstring) from one program to the next, things get a bit more involved. I will try to spare you the messy details as much as possible, but I must cover a few points to explain why C-64 chaining involves what it does.

First, as you are aware, your C-64 has several kilobytes of working memory area. Different parts of this area are assigned to hold different information. Usually, a programmer can leave it to the operating system to worry about what is located where, and simply ignore it. Unfortunately, your C-64's operating system lacks the smarts to manage things properly when you attempt to link two programs together and pass the variables. It is pretty smart, but not that smart. If you try to make it do this job for you, you find that it has reached its "level of incompetence," to use the term coined by an author named Peter a few years ago.

When you LOAD a program, it is assigned a particular range within your program's memory. When you later enter data—variables, arrays, strings—they are assigned to areas of memory above the program. As long as the program stays put, everything is fine. If you LOAD a new, larger program, however, it may extend up to the areas reserved for variables, arrays, and strings, thereby destroying them. This is what happened in the little exercise with Program A and Program B.

The solution comes in two steps. First, protect the variables and arrays. Second, protect the strings.

All that is required to protect the variables and arrays is a very simple trick. As you saw, in chaining from Program A to Program B, Program B overwrote the memory area containing variables, arrays, and strings. Suppose that Program A were bigger than Program B? In that case, Program B would no longer extend into the wrong memory area.

The problem encountered earlier in attempting to chain from Program A to Program B was caused by Program B being slightly bigger than Program A. You can fix part of the program by changing Program A to the form shown in Fig. 7-5. This program (A') is identical to Program A except for lines 70 and 80—which do nothing more than add a few bytes to Program A', thereby making it bigger than Program B. Modify Program A to look like Fig. 7-5 and then SAVE it to disk as PROGRAM A'.

Now RUN Program A' and see what happens.

This time the values assigned to the real variable and array make the trip successfully to Program B, but the strings do not make it, and the character color changes because of some string confusions. Still, this exercise illustrates that you can pass the values of variables and arrays if Program A' is bigger than Program B.

I am not suggesting that you add dummy lines to your first program to make it bigger than the second. Although the example just given illustrates the idea, doing it this way is not very efficient. Besides, there is another way to do it. Your C-64 measures the size of a program when it LOADs it, and stores this value in memory. It also decides where to put the variables, arrays, and strings, and stores these in memory. The respective values are stored as two bytes each in "low-byte high-byte" format. If you are unfamiliar with this, the following formula allows you to compute the decimal value from a low-byte and high-byte pair:

$$\text{Decimal Value} = \text{LO} + 256 * \text{HI}$$

For example, if LO is 150 and HI is 15, then Decimal Value = 150 + 15 * 256, or 3990. As a practical

```
10 REM PROGRAM B
20 PRINT"A=";A
30 PRINT"A(5)=";A(5)
40 PRINT"A$=";A$
50 PRINT"A$(5)=";A$(5)
60 REM**********************************
70 REM**********************************
```

Fig. 7-4. Program B (bigger than Program A).

```
10 REM PROGRAM A'
20 A=100
30 A(5)=555.55
40 A$="A STRING"
50 A$(5)="A STRING OF FIVE"
60 LOAD"PROGRAM B",8
70 REM**********************************
80 REM**********************************
```

Fig. 7-5. Program A' (bigger than Program B).

matter, the LO byte contributes much less to the total and can be safely ignored if you add a safety factor—1 or 2—to the HI byte.

Your C-64 uses the memory locations shown in Table 7-3 to store several pointers that are important during chaining. When your C-64 executes a program, it allocates various memory areas to the variables, arrays, and strings, and stores the pointers to the memory areas in locations 45 through 52. The allocated areas are always above the program itself. Now, if you LOAD your largest program, and PEEK at the values of memory locations 45 and 46, you can see where it has allocated the lowest of these areas in the largest program. Knowing this value, you can take it back to your first program and POKE the value in there, and thereby do the allocating yourself instead of letting your C-64 do it.

To illustrate the use of the PEEKs and POKEs, LOAD Program B and type in and record the value you obtain by PRINTing the following statement:

PEEK (46)

You should get 8. Now add 1 to this value (9), and add lines 11-14 to the original Program A shown in Fig. 7-3. The new program (Program A") is shown in Fig. 7-6. Save this program to disk as Program A". Then RUN it and see what you get when Program B executes.

What should happen is that the real variable and array survive, but the strings, once again, get lost during the trip.

The procedure just described, in summary, is as follows:

- LOAD the largest subprogram.
- PEEK memory location 46.
- Add 1.
- Add lines at the start of the first program that POKE the resulting value to memory locations 46, 48, 50, and 52.

When you POKE the value into locations 46, 48, 50, and 52, you are telling the computer that the POKEd value is where the top of the subroutine is located, and it can begin putting variables, arrays, and strings above that. It does not argue, but follows your order obligingly, allowing itself to be tricked.

A word of caution. Be careful how you SAVE a subprogram that assigns its own values to memory locations 46 through 52. Your C-64 is not smart

Pointer	Low Byte	High Byte
Start of Variables	45	46
Start of Arrays	47	48
End of Arrays	49	50
String Storage	51	52

Table 7-3. Background and Character Color Codes.

```
10 REM PROGRAM A''
11 POKE 46,9
12 POKE 48,9
13 POKE 50,9
14 POKE 52,9
20 A=100
30 A(5)=555.55
40 A$="A STRING"
50 A$(5)="A STRING OF FIVE"
60 LOAD"PROGRAM B",8
```

Fig. 7-6. Program A" (smaller than Program B but with POKEd size pointers).

enough to know that you have tricked it. If your subprogram only takes two K bytes of memory, but you have tricked your C-64 into thinking it takes up 22 K, it allots 22K of disk space when it SAVEs the subprogram. You can waste a lot of disk space this way. Do not ever SAVE such a program immediately after you have RUN it. Clear memory first and LOAD the subprogram from scratch. This process resets the memory pointers and allows the program to be SAVEd in the amount of space that it actually needs.

Transferring Strings and String Arrays

Your C-64 does not have as much respect for strings as it does for variables. They probably know how Rodney Dangerfield feels. Although an area of memory is allocated to string storage, not every string receives the call, and most of them go through the meat grinder when you chain. The reason is that BASIC leaves strings in the program area, rather than storing them in a separate area—unless the program does certain things to them. You can be sure that strings receive the respect they deserve by concatenating all strings with the null string. For example, Set A$ = A$ + "".

To illustrate how this works, Fig. 7-7 is a modified version of Fig. 7-6 in which lines 45 and 55 have been added to concatenate the strings and string array element with the null string. Type in these two lines to Program A", and then SAVE the program to disk as Program A'".

Run it and see what happens. If all worked according to plan, both variables and string should have found their way safely to Program B.

The foregoing illustrates that it is possible to preserve the identity of strings by concatenation. It follows from this that if you do not concatenate strings, they are not preserved. A string that is "not preserved" does not turn into the null string, but may take on an astonishing shape if you attempt to use it later on. If you have a color monitor and have been faithfully following the keyboard exercises described in this chapter, then you probably saw the character color of your display change when some nonpreserved strings were printed during Program B. This and stranger things can happen. The lesson is that, if you choose not to preserve a string, do not attempt to use it later on unless you first redefine it. If you use it without doing so, you may be in for some interesting surprises.

Chaining Summary

Just in case you got lost in the foregoing, or it told you more than you wanted to know, here is a summary of what is important:

- LOAD the largest subprogram.
- PEEK memory location 46.
- Add 1 to the value you find (call the resulting value X).
- Put these lines at the beginning of your first subprogram:

```
10  POKE 46,X
20  POKE 48,X
```

```
10 REM PROGRAM A'''
11 POKE 46,9
12 POKE 48,9
13 POKE 50,9
14 POKE 52,9
20 A=100
30 A(5)=555.55
40 A$="A STRING"
45 A$=A$+""
50 A$(5)="A STRING OF FIVE"
55 A$(5)=A$(5)+""
60 LOAD"PROGRAM B",8
```

Fig. 7-7. Program A'" (Program A" with string concatenation).

```
30  POKE 50,X
40  POKE 52,X
```

(use the actual number that X represents (such as 10), not the variable X.)

- To protect strings during chaining, concatenate them with the null string or READ them in from DATA statements.
- Warnings:
  - —After chaining, do not use strings that were unprotected during chaining without first redefining them.
  - — After executing a program with the lines just given, do not SAVE it without first clearing memory completely and LOADing the program from scratch.

That is all there is to it! Well, I never said it was easy. It is also straightforward to remove your own appendix, if you know what you are doing. Look at it this way: if it was easy to chain, then everyone would know how, and where would that leave us? Better to sustain a few mysteries.

If you buy that, have I got a great deal for you on the Brooklyn Bridge . . .

If you want to feel really bad about this, be aware that, in some BASICs, the only thing you have to do to chain from one program to the next is insert a line that says CHAIN "NEXT PROGRAM".

Well, perhaps one day Commodore will design a machine that way. Until then, keep your bag of tricks handy.

PROGRAM VERIFICATION

The chaining technique just described works fine if the program being chained to is present on disk when your drive goes looking for it. If it is not there, then a disk error occurs, and your system crashes.

You will not have this problem if all of your subprograms are on one disk that is never removed from your drive. In that case, there is never a danger of removing the disk that contains the subprogram that your drive is looking for.

It gets trickier, however, if your program requires disk swapping. Disk swapping, as you know, is the popular name for putting one disk in your drive for a while, pulling it out, putting another in, putting the first back in, and so on. Disk swapping is required when all of the files required by your program are not on the same disk. Your subprograms may take up too much space to fit on one disk, for example. This is fairly unlikely for most of the types of programs written for the C-64. What is more likely is that you will have your subprograms on a program disk and your data files on a data disk. If you have one disk drive, this means that you must make sure your program disk is in the drive when chaining is about to occur, and that your data disk is in the drive when a data file must be read or written to. If you have two disk drives, things are easier. Then you can put your program disk in one drive and your data disk in another.

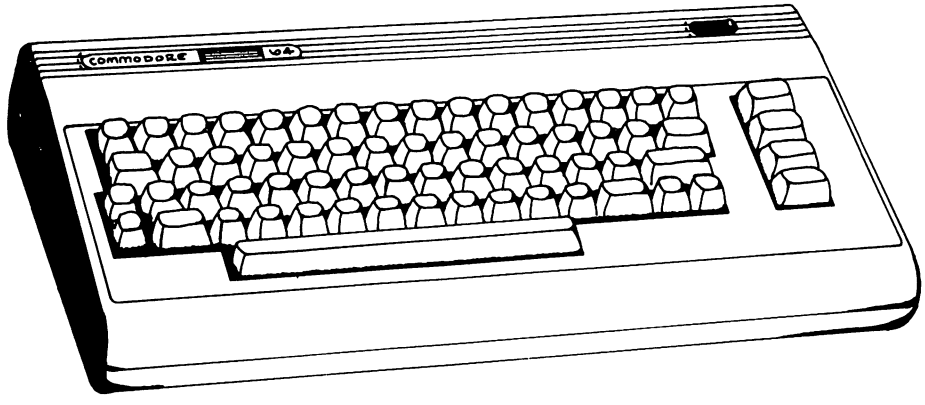
Most C-64 owners have one disk drive, rather than two. Many serious programs require data files. Put these two facts together and they add up to a good chance that disk swapping is necessary. Take that a step further and you reach the conclusion that problems can easily occur.

There is a solution, however. It has four parts:

- Your computer must keep track of which disk is in the drive at any given point in time.
- Your program must prompt the user to make disk swaps, as necessary.
- Your program should not attempt to read or write a file until the user has verified that the proper disk is in the drive.
- Your program should not trust the user to verify the disk correctly, but must verify the disk itself.

Chapter 8 (File Handling) describes ways in which these four functions can be managed using C-64 BASIC.

Chapter 8



File Handling

This chapter concerns disk files. Its primary focus is on data files; these are the files that the serious programmer uses to store data entered by the user through the keyboard. Later on, the files are read back in and the user's earlier entries come back to life. Many, perhaps most, serious programs use data files. It is important to know how to handle them properly, efficiently, and with minimum complications.

There is a good deal more to file handling than simply being able to create a file, write to it, and then read it back into memory. This is the heart of the matter, but not enough to allow you to write a program that handles files safely or well. You also need to be able to plan your files intelligently, deal with error conditions, prompt the user to make necessary disk swaps, and design a control structure to read and write the files efficiently. This chapter covers these subjects. Knowing them makes your life as a programmer easier and allows you to write programs that are safe for users.

As mentioned in Chapter 2, the C-64 has fairly limited error-handling capabilities. Unlike many microcomputers, it lacks an ON ERROR GOTO statement that allows a programmer to trap errors which occur during a program to prevent the program from crashing or locking. This makes using any C-64 program like walking a tightrope without a net. A skilled tightrope walker can do it, although it takes care and practice. Similarly, you can do it—make sure you do not have any syntax errors, GOTOs to nonexistent lines, data-entry routines with holes in them, and so forth. You can, in fact, make the case that most of the types of errors that an ON ERROR GOTO statement protects against are ones that the programmer should prevent in the first place.

There are, however, some errors that no programmer, no matter how careful, can protect against. For example, if the program user has the wrong disk in the drive when the program attempts to read a particular file, an error condition occurs. Commodore, perhaps recognizing such human error possibilities, has provided a way of trapping read/write errors. It is done by reading the notorious error channel. Doing this is cumbersome, but it

does work, and, as the last woman on earth said to the last man, "You may not like me, but I'm all you've got."

You can get a bit of additional mileage out of the error channel. By doing a variation on the technique, you can put a safety net beneath two programs that are chained together. Chapter 7 showed how to chain, but not how to verify that the correct disk is in the drive before chaining. It could not cover verification because this subject requires reading the error channel, a subject not covered until this chapter. The remainder of that discussion is contained later in this chapter, under "File Verification."

Once you have mastered the techniques for reading and writing files, reading the error channel, dealing with disk swaps, and so forth, you may forget a lot of what you worked so hard to learn. You can become quite competent at writing these routines if you work with them often enough. If you do not, then you go through the normal process of learning decay, which is the fancy term for forgetting. Then, when you want to write a new program, you must learn everything all over again.

There is, of course, an easier way: use subroutines. I played this tune before, and there is nothing new about it. If I could, however, I would turn the volume up a few notches for this chapter. The file read and write syntax is especially confusing, and that is why it is especially important to use subroutines to deal with files. Once you have a good set of these in your library, you can pull them out and adapt them to a new application. It makes life much easier.

There is another aspect to this, as well. If you have ever attempted to translate a program from one dialect of BASIC to another—say, from Apple II BASIC to C-64 BASIC—then you know what an adventure it can be. The difficulty of translating is influenced a good deal by how the program was first designed. If it was built around subroutines, translation is much easier. The reason for this is not hard to understand. The control code of such a program is at a fairly high level. Much of it consists of arguments and subroutine calls. These are very similar from BASIC to BASIC. That is, a GOSUB 3000 means the same in virtually any dialect of BASIC. In short, using subroutines makes it easier to translate your program.

This chapter covers several different topics. It begins with a discussion of file planning—how to decide what types of files you need, and what goes in them, how much memory they require, and how to document them. The second section covers the fundamentals of writing and reading files. It shows how to develop core subroutines to write and read sequential files, random-access files, and pseudorandom-access files. The third section tells how to read the error channel and incorporate an error-handling routine in your file write and read subroutines. The fourth section covers file verification, and the fifth section covers management of disk swapping with one-drive systems. The final section tells how to make your file write and read code both efficient and safe—by building in enough intelligence to ensure that files are written or read only when necessary, but often enough that no data is lost.

FILE PLANNING

It is very important to plan your files carefully before you start writing code. If you do not do this properly, you may find yourself modifying their structure, adding new ones, deleting others, and being in a state of confusion. During the development of a program that uses files, you need the code to write and read them really early. Much of the early coding work makes assumptions about those files. Consequently, changes to files have a ripple effect throughout the design and have significant time, work, and error consequences.

Plan carefully, and do it early. This section identifies and discusses the key factors to consider during planning.

Decide What Types of Files You Need

There are two types of files: sequential, and random access. A *sequential file* has one record and is loaded into memory or written out to disk all at once. Sequential files are commonly used for storing arrays. A sequential file can, however, be as short as one character, and in fact such files come in

handy for certain applications. The prime characteristic of a sequential file is not its length or what is stored in it, but that its entire contents are handled at one time. When your program reads it, it reads it all at once. When it writes it, it writes it all at once. Sequential files are useful for storing directories, indexes, and other information that can be considered (more or less) one of a kind. If your program must deal with a number of different data sets, however, then it is better to use a random-access file.

A *random-access file* is like a card file. It consists of records, and each record has its own unique record number. You can read or write one record without reading or writing the entire file. Random-access files are required when your program must deal with several different data sets—for example, identically formatted personnel records of different people, price histories of different stocks, or market analyses of different real estate properties.

The C-64 can use two different types of random-access files. These may, for the sake of discussion, be called Type 1 and Type 2. Type 1 files require you to worry about blocks, sectors, tracks, pointers, and other matters that most computers leave to their operating systems, although these files can be handy if you do machine-language programming. Type 2 files are much easier to work with since they are similar to standard BASIC random-access files. This book covers Type 2 files, but not Type 1 files. Most serious programs can be written very nicely with Type 2 files, and these files are also much easier to use.

There is no upper length limit for a sequential file. You can make it as long as you want, provided you have room in memory to store its contents, space on disk to hold it, and the patience to wait for your disk drive to write or read the file. Random-access files are a different matter. The C-64's DOS limits you to a maximum of 720 records, and no record can be more than 254 characters in length. These are fairly short records, although you can have a respectable number of them. This length limitation is one of the reasons to plan your records carefully—if they are too long you cannot use a standard random-access file.

You can use a set of sequential files to simulate a record-oriented file. Since sequential files have no length limitation, you can, in effect, create a pseudorandom-access file that has records longer than those possible with a standard random-access file. The technique is described later in this chapter.

Document Your Files

Once you have decided what types of files you need—sequential or random access—lay them out on paper. It is not difficult to write a file-handling subroutine (one for writing or reading a file), but you must take many details into account. You should not worry about these details when you sit down before your computer to write the code. Sort them out beforehand. The actual writing of the file-handling code should be as mechanical a task as possible. You should also be able to base your new routines on a standard formula. Writing a program is not like writing a novel. Instant inspiration is more a hazard than a blessing. Systematic, step-by-step working habits get you to the goal more quickly, and with fewer errors, than the programmer's muse.

The best way to plan your files beforehand is to document them. A suggested format for doing this is shown in Fig. 8-1. You can use this format for either sequential or random-access files. Documentation headings are as follows:

- Type of file—This will be either random access or sequential.
- File Name—The name you attach to the file, such as NAME, DATA INDEX, and so forth.
- Flag—The flag used in write subroutines to govern whether or not a file is written. The use of flags is discussed later in this chapter.
- File Number—In addition to a name, each file must have a number that is used in its write and read subroutines. File numbers must be between 2 and 14. If your program uses several files, it is a good idea to follow

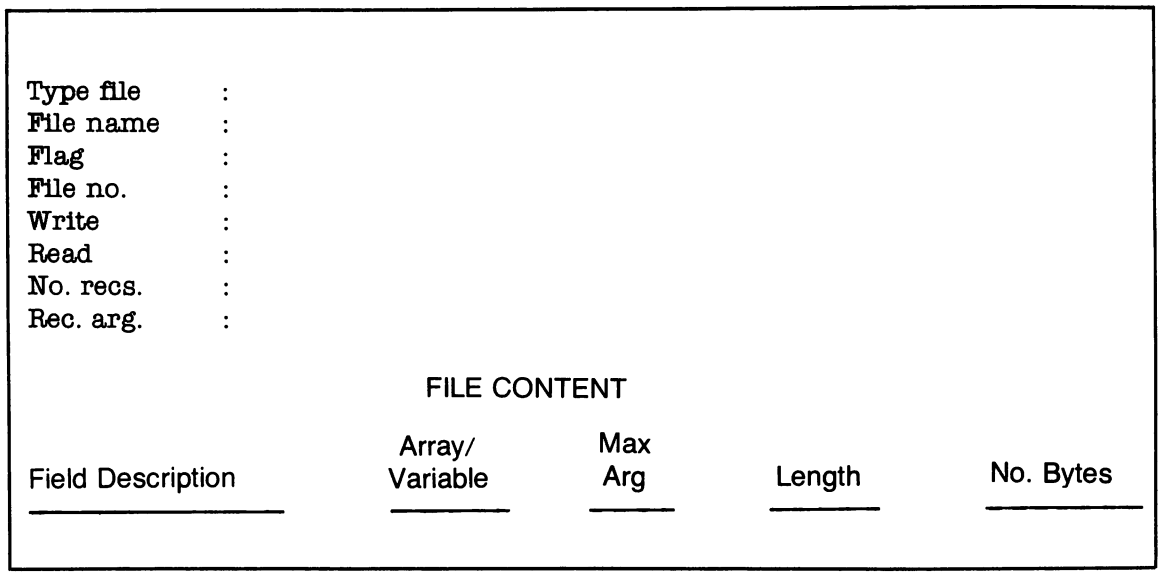


Fig. 8-1. Suggested format for documenting files.

a rule for numbering the files. For example, use file numbers 2-5 for sequential files, 6-10 for standard random-access files, and 11-14 for pseudorandom-access files.

- Write Line Number—The program line number at which the file write subroutine begins.
- Read Line Number—This is the program line number at which the file read subroutine begins.
- Number of Records—If a random-access file, the number of records it contains.
- Record Argument—If a random-access file, the variable used for setting the record number. The most common variable is probably R.
- Field description—What each field in the file contains. For example, a field may contain someone's name, a price, and so on.
- Array/Variable—How the field is represented as an array or variable. For example, a name may be represented as NAME$, price as PR, and so on.
- Maximum Argument—If an array is used, this is the number of elements in the array (the dimensioned value). Otherwise this value is 1.
- Length—The length, in characters, of the field.
- Number of Bytes—This is the number of characters the field requires on disk. How much space is needed depends upon the type of field—string, real, integer—and what goes in that field. Follow these guidelines:

1. String fields take as many bytes as they have characters, such as "A"(1 byte), "ABLE" (4 bytes), "MISANTHROPE" (11 bytes). Add one byte to each for the field separator (,).
2. Real fields take from 2 to 15 bytes, depending on the number of characters. You must add one for an extra space that DOS inserts at the end of the number during storage. Examples: 1 (2 bytes), 3.5 (4 bytes), 1.23456789 (11 bytes), 3.34567893%+20 (15 bytes). Again, add another byte to each field for the separator.
3. Integer fields take from 2 to 7 bytes, depending on the number of characters. You must add one for an extra space that DOS inserts at the end of the number during stor-

age. Examples: 2 (2 bytes), −45 (4 bytes), 32767 (6 bytes), −32768 (7 bytes). Add another byte for the field separator.

4. After determining the Maximum Argument and field Length, use the following formula to compute the number of bytes: No. Bytes = Max. Arg * (Length + 1) (The extra 1 accounts for the field separator). For example, if a 10-element string array is used, and each field takes 24 bytes. No. Bytes = 10 * (24 + 1) = 250 bytes. Note that, if element 0 of an array is used, the number of bytes = (Max. Arg + 1) * (Length + 1).

You can use Fig. 8-1 as a framework for preparing file documentation for a specific program. Figures 8-2 and 8-3 contain such documentation for a sequential and a random-access file, respectively. The content of these figures is self-evident—I leave it to your common sense to sort them out.

In the next section, I will develop subroutines for writing and reading these two files. Note that, once you document a file in this form, you can use the documentation as a specification for creating write and read subroutines.

DESIGNING SUBROUTINES TO WRITE AND READ FILES

This section tells how to write the code of

FILE DOCUMENTATION

Type File : Sequential
File name : SEQFILE
Flag : —
File no. : 2
Write : 6010
Read : 6250
No. recs. : —
Rec. arg. : —

FILE CONTENT

Field Description	Array/ Variable	Max Arg	Length	No. Bytes
Names	A$(A)	5	12	65
Prices	A(A)	5	11	60
Ages	A%(A)	5	7	40
Proctor	A$	1	12	13
Fee	A0	1	11	12
Identification No.	A%	1	7	8
Total = Max Arg (Ltd)			Total	198

Fig. 8-2. Documentation for hypothetical sequential file.

FILE DOCUMENTATION

Type file : Random
File name : RANDFILE
Flag : —
File no. : 6
Write : 7050
Read : 7280
No. recs. : 10
Rec. arg. : R

FILE CONTENT

Field Description	Array/ Variable	Max Arg	Length	No. Bytes
Names	A$(A)	5	12	65
Prices	A(A)	5	11	60
Ages	A%(A)	5	7	40
Proctor	A$	1	12	13
Fee	A0	1	11	12
Identification No.	A%	1	7	8
			Total	198

Fig. 8-3. Documentation for hypothetical random-access file.

subroutines that write and read sequential and random-access files. It also shows how to use sequential files as if they were random-access files, a technique that comes in handy if standard random-access files are too limiting for your application. The techniques illustrated should take care of most, if not all, of your needs in any serious program you write.

Sequential Files

In creating write and read subroutines, follow these conventions:

- File-handling subroutines use line numbers 6000 through 8499. (Incidentally, related subroutines, for error-handling and disk swap management, use the latter part of this range.)
- The write subroutine always precedes the read subroutine.
- There is a line increment of 250 (more or less) between write and read subroutine.

There is nothing magical about these rules, but they help keep things straight. Write and read subroutines are always adjacent, and their line numbers are not far apart. In working with one, you are never far from the other.

```
6010 REM--SEQUENTIAL FILE (WRITE)--
6020 OPEN 2,8,2,"@0:SEQFILE,SEQ,W":REM OPEN FILE
6030 REM-STRING ARRAY-
6040 FOR A=0 TO 5
6050 PRINT#2,A$(A)
6060 NEXT
6070 REM-REAL ARRAY-
6080 FOR A=0 TO 5
6090 PRINT#2,A(A)
6100 NEXT
6110 REM-INTEGER ARRAY-
6120 FOR A=0 TO 5
6130 PRINT#2,A%(A)
6140 NEXT
6150 REM-VARIABLES-
6160 PRINT#2,A$
6170 PRINT#2,A0
6180 PRINT#2,A%
6190 CLOSE 2:REM CLOSE FILE
6200 RETURN
```

Fig. 8-4. Subroutine to write or create sequential file.

Let us create subroutines to write and read the sequential file documented in Fig. 8-2, starting with the write subroutine. This file contains three 11-element (counting 0) arrays—string, real, and integer—and one string, real variable, and integer variable. This is probably not a very realistic file, but it contains the building blocks for many that are. Its contents can be modified, as necessary, for use in actual programs.

Writing. Figure 8-4 contains the code for creating and writing the file documented in Fig. 8-2. The write subroutine consists of lines 6010-6200. Let us go through this subroutine line by line.

Line 6020 contains the file OPEN statement. The syntax of this statement is as follows:

OPEN File Number, Device Number, Channel Number, "@0:File Name,SEQ,W"

File Number and Channel Number do not have to be the same, but it simplifies things to make them so. Think of them both as File Number, and forget about Channel Number. They can be any numbers between 2 and 14. (I suggest you use numbers 2 through 5 for sequential files.) Figure 8-4 uses File and Channel Numbers of 2.

Device Number is 8 if the disk is in drive 1, and 9 if it is drive 2. Figure 8-4 uses Device Number 8.

File Name is whatever you choose to call the file—it is SEQFILE in Fig. 8-4.

SEQ stands for SEQuential. W means Write (or use R for Read). Put all this together and you come up with line 6020. So much for OPENing the file. From here on it gets easier.

Lines 6040-6180 contain PRINT# statements that PRINT the data into the file. The number following the PRINT# statement is the Channel Number (2), as used in the OPEN statement on line 6020.

Lines 6040-6140 print three string, real, and integer arrays to the file. Each array is printed with a FOR-NEXT loop. Each element of the array can be printed separately, but this is not as efficient as using a FOR-NEXT loop.

Lines 6150-6180 print string, real, and integer variables to the file. Each item must have its own PRINT# statement, as in line 6160. It is possible to print several items to the file with a single PRINT#

statement by separating the items with commas. For example, you can print all three items with this statement:

```
PRINT#2,A$,A0,A%
```

Do not do this. The reason is that, when DOS prints this to the file, it separates each item by eight spaces, just as if printing information to separate fields on your display. This leaves a lot of extra, wasted space in the file.

Line 6190 contains the CLOSE statement, which CLOSES the file. Line 6200 contains the RETURN, which marks the end of the subroutine.

How do we use this subroutine? The answer is easy. Right? When we want to create the file, we make sure that arrays have been dimensioned and then just give a GOSUB 6010. Right? Almost. There is one additional requirement.

DOS has a problem with null strings—strings whose content has not yet been defined. To illustrate, if you write an undefined string to a file, DOS acts like nothing is there. It does not put a separator (,) between the null string and the next item. This means that the file is not properly created. The most interesting things happen when you attempt to create a file containing a string array and do not define all of the elements of the array before writing it to the file. For example, suppose that you define the elements of the string array A$ as follows:

```
A$(3)="3"
A$(4)="4"
A$(5)="5"
```

(Assume that A$(0) through A$(2) are not defined.) Now write this array to file, and then read it back in and PRINT each array element. Here is what you get:

```
A$(0)=3
A$(1)=4
A$(2)=5
A$(3)=
A$(4)=
A$(5)=
```

All of the strings you assigned slipped down the array, because the null strings that were printed there collapsed like weak gymnasts beneath the weight of their more substantial teammates. In short, if you leave any string array elements undefined, those elements do not write to the file, and your file is improperly created.

The simplest solution to this problem is to define every element of the array. You may not know what is supposed to go there when you create the file, but put something there anyway. Just about anything will do. I suggest the hyphen or some other seldom-used character that is easy to test for later on when defining the array.

Note that DOS does not have this problem with real or integer arrays, and that you do not have to assign their values before writing the file.

That covers the creation and writing of sequential files. I will add some additional features later in the chapter, but what I have just discussed is the core of all such routines. It has three basic parts: OPEN file, PRINT# data to it, and CLOSE file.

Reading. The procedure for READing a sequential file also has three parts: OPEN file, INPUT# data, and CLOSE file. The parts for writing and reading a file are very similar—in fact, enough so that any programmer with common sense first creates the write subroutine and then uses the C-64's line editor to convert it into a read subroutine. More on this later.

Figure 8-5 contains a subroutine for reading the sequential file created by the subroutine in Fig. 8-4.

Line 6260 OPENs the file for reading. This line is identical to the OPEN file statement used in the write subroutine except that the last character is R (for Read) instead of W (for Write).

Lines 6280-6420 read in the data from the file with INPUT# statements. The INPUT# statement is analogous to the PRINT# statement used in the write routine. In using the INPUT# statement, however, no harm is done by INPUTing several variables on the same line, separated by commas. The number following the INPUT# statement is the Channel Number, as used in the OPEN statement on line 6260.

Line 6430 CLOSEs the file. Line 6440 ends the subroutine with a RETURN statement.

Converting a Write Subroutine to a Read Subroutine. It is easy to convert a write subroutine to a read subroutine. There are two basic approaches. The first is to copy the write subroutine alone to a separate file, renumber it, convert it, and then merge it with the original program.

The second approach is to work within the same program by going through the write subroutine line by line, changing line numbers and producing a duplicate of the write subroutine at a different line range. Either way works and is faster than creating the second subroutine from scratch.

Once you have renumbered the write subroutine, make the following changes:

- Change the W at the end of the OPEN statement to R.
- Change all PRINT# statements to INPUT# statements.
- Optionally, you can remove separate PRINT# statements for individual strings and variables and list them on one line, separated by commas.

Try It Out. Figure 8-6 contains the listing of some code that creates a dummy data set, writes it to SEQFILE, and then reads it back in and displays it.

Lines 1000-10100 create the dummy data set. Line 10110 calls the file-write subroutine, which creates SEQUFILE. Line 10120 then clears all variables. Line 10130 calls the file-read subroutine, which reads the data set back in. Lines 10150-10180 display the contents of the file.

Create a program consisting of Figs. 8-4 through 8-6. This gives you an opportunity to try these subroutines out and to manipulate the data set and observe the effects when the file is read back in.

Be sure to set some of the intermediate elements of the string array to the null string and see what happens. The read subroutine should crash; when it does, PRINT the contents of the array and check the values you assigned in lines 10020-10100.

Random-Access Files

There are three steps in building a random-access file:

- Plan the record.

```
6250 REM--SEQUENTIAL FILE (READ)--
6260 OPEN 2,8,2,"@0:SEQFILE,SEQ,R":REM OPEN FILE
6270 REM-STRING ARRAY-
6280 FOR A=0 TO 5
6290 INPUT#2,A$(A)
6300 NEXT
6310 REM-REAL ARRAY-
6320 FOR A=0 TO 5
6330 INPUT#2,A(A)
6340 NEXT
6350 REM-INTEGER ARRAY-
6360 FOR A=0 TO 5
6370 INPUT#2,A%(A)
6380 NEXT
6390 REM-VARIABLES-
6400 INPUT#2,A$
6410 INPUT#2,A0
6420 INPUT#2,A%
6430 CLOSE 2:REM CLOSE FILE
6440 RETURN
```

Fig. 8-5. Subroutine to read sequential file.

```
10000 REM--TEST FILES--
10010 REM-CREATE DUMMY DATA SET-
10020 DIM A$(5),A(5),A%(5)
10030 FOR A=0 TO 5
10040 A$(A)=STR$(A)
10050 A(A)=A
10060 A%(A)=A
10070 NEXT
10080 A$="MYNAME"
10090 A0=1.23456789
10100 A%=-32768
10110 GOSUB 6000:REM WRITE FILE
10120 CLR:REM CLEAR MEMORY
10130 GOSUB 6250:REM READ FILE
10140 REM-DISPLAY FILE CONTENTS-
10150 FOR A=0 TO 5
10160 PRINT A$(A),A(A),A%(A)
10170 NEXT
10180 PRINT A$,A0,A%
```

Fig. 8-6. Program code to create dummy data set and write and read SEQFILE.

- Code the write and read subroutines.
- Create the file.

Record Planning. It is especially important to plan random-access files because a record cannot be longer than 254 characters, and you cannot have more than 720 records in the file. If your program attempts to create or write a record longer than 254 characters, or to use a record number higher than 720, an error condition occurs.

Use a record-planning form such as that shown in Fig. 8-1 to lay out your record and determine its length. To illustrate, suppose that you want to create a random-access file with 10 records to hold the information you were storing on disk with the sequential file documented in Fig. 8-2. Figure 8-3 documents this random-access file. Documenting a random-access file is almost identical to documenting a sequential file; the only additional requirement for the random-access file is information concerning Number of Records and Record Argument.

Let us briefly review Fig. 8-3. Type File is Random. File Name is RANDFILE. File Number is 6; I suggest that you use identical File and Channel Numbers of 6 through 10 for random-access files. The write subroutine starts at line 7000 and the read subroutine at 7250. There are 10 records, and the Record Argument is R.

File Content is identical to that of the sequential file created earlier, although the random-access file will contain 10 records with this content. The Maximum Argument, Length, and Number of Bytes are computed, and their total memory requirement (198 bytes) is shown at the bottom right of the figure—this is how long each record must be.

Creating the File. To create RANDFILE, you must insert specific lines of code in your program to perform the act of file creation. You cannot simply write to the file; you can create a sequential file by writing to it, but a random-access file is more demanding.

The syntax of the code required to create a random-access file is as follows:

OPEN File Number, Device Number, Channel Number, "File Name,L,"+CHR$(Record Length)

Let us make File Number and Channel Number identical, and set them to 6. Device Number for a single drive is 8. File name is RANDFILE." Record length is 198. The code required to create the file is as follows:

```
20000 OPEN 6,8,6,"RANDFILE,L"+CHR$(198)
```

There is no point in putting this code into a subroutine since it is only executed once. Locate it in your program where the file is to be created. For example, to create the file at line 20000, include these lines in your program:

```
20000   OPEN 6,8,6"RANDFILE,L"+CHR$(198)
20010   CLOSE 6
```

After these lines have been executed, the file exists, and records may be written to it. Do not allow the program to attempt to read a record that has not been written since the record is incomplete, and an error condition can occur. It is a good idea to

write out every record, in turn, immediately after creating the file. As always, be sure to define all strings as something other than the null string.

To illustrate, let us create records 0-11 of RANDFILE. The following code will create the file, define the strings, and write these 11 records to the file:

```
20000 OPEN 6,8,6,"RANDFILE,L," + CHR$
(198)
20010 CLOSE 6
20020 REM—DEFINE STRINGS—
20030 FOR A=0 TO 5
20040 A$(A)="—"
20050 NEXT
20060 A$="—"
20070 REM—WRITE RECORD TO FILE—
20080 FOR R=0 TO 10
20090 GOSUB 7000:REM CALL FILE
WRITE SUBROUTINE
20100 NEXT
```

Write and Read Subroutines. The code for writing and reading a random-access file has some similarities to that for a sequential file, and also some differences. In both cases, code is required to OPEN and CLOSE the file at the start or end of a write or read operation. For a random-access file, however, two channels must be opened—the command channel, and the channel to transmit the file data through. The syntax of the file OPEN statement for a random-access file is simpler than that for a sequential file—it is shorter, does not require an R (read) or W (write) code, and is identical for both write and read operations. Finally, the PRINT# statements and INPUT# statements are identical in form for both sequential and random-access files.

Three lines are required to OPEN the command channel, OPEN the file, and position the file pointer to read the required record.

The syntax of the statement for OPENing the command channel is:

OPEN File Number, Device Number, Channel Number.

I suggest that you use 15 for File Number. Device Number is 8 for drive 1 or 9 for drive 2. Thus, using drive 1, this statement OPENs the command channel:

```
OPEN 15,8,15
```

The syntax of the statement for OPENing the file is:

OPEN File Number, Drive Number, Channel Number, "File Name"

Assume File and Channel Numbers of 6 and a File Name of RANDFILE. This statement will OPEN the file:

```
OPEN 6,8,6 "RANDFILE"
```

The tricky part, actually, is to position the file pointer—locate the record to be read. The syntax of this statement is as follows:

PRINT#15,"P"CHR$ (Channel Number)CHR$ (Rec.No. Lo byte)CHR$(Rec.No.Hi byte)

Since the file contains many records, a statement (this one) must identify the record of interest. In this statement, the command channel (15) is used to direct DOS to the desired record. The "P" stands for Position; that is, the position the file pointer must go to read the record.

CHR$ (Channel Number) identifies the channel number. In our example, Channel Number is the same as File Number (6).

The next two CHR$ terms identify the record number in low-byte and high-byte form (as discussed in Chapter 7). The actual record number, R, is computed from REC.No.Lo (LR) and Rec.No.Hi (RH) according to this formula:

$$R = RL + 256 * RH$$

This is not the easiest way to indicate Record Number, but it does work. To illustrate how, let us consider a few examples.

To find record number 50, use this statement:

```
7000 REM--RANDOM FILE (WRITE)--
7010 GOSUB 8000:REM CALCULATE RL & RH
7020 OPEN 15,8,15
7030 OPEN 6,8,6,"RANDFILE"
7040 PRINT#15,"P"CHR$(6)CHR$(RL)CHR$(RH)
7050 REM-STRING ARRAY-
7060 FOR A=0 TO 5
7070 PRINT#6,A$(A)
7080 NEXT
7090 REM-REAL ARRAY-
7100 FOR A=0 TO 5
7110 PRINT#6,A(A)
7120 NEXT
7130 REM-INTEGER ARRAY-
7140 FOR A=0 TO 5
7150 PRINT#6,A%(A)
7160 NEXT
7170 REM-VARIABLES-
7180 PRINT#6,A$
7190 PRINT#6,A0
7200 PRINT#6,A%
7210 CLOSE 6
7220 CLOSE 15
7230 RETURN
```

Fig. 8-7. Subroutine to write random-access file.

PRINT #15,"P"CHR$(6)CHR$(50)CHR$(0)

To find record number 257, use this statement:

PRINT #15,"P"CHR$(6)CHR$(1)CHR$(1)

To find record number 720, use this statement:

PRINT #15,"P"CHR$(6)CHR$(208)CHR$(2)

If your file has fewer than 256 records, then forget about RH and set RL to R. That is, use this kind of statement in your code:

PRINT #15,"P"CHR$(6)CHR$(R)CHR$(0)

If your file has more than 255 records, then you must provide both arguments, RL and RH. It is much easier to use one record number argument (R) than two—especially for users. You certainly cannot expect them to type in which record they want in low-byte and high-byte form. The easiest way to calculate RL and RH is with a subroutine that converts one record number argument, R, to the two arguments, RL and RH. I will show how to do this presently, but first let me show how to OPEN the command channel, OPEN the file, and position the file pointer in an actual file write subroutine.

Assume that you want to locate record number 10 in RANDFILE. These lines will OPEN the file and position the pointer as required:

```
10 OPEN 15,8,15:REM OPEN COMMAND CHANNEL
20 OPEN 6,8,6:REM OPEN FILE
30 PRINT#15"P"CHR$(6)CHR$(10)CHR$(1):REM POSITION POINTER
```

After OPENing the file in this manner, and

positioning the record pointer, data may either be written to the file with PRINT# statements or read from the file with INPUT# statements. After so doing, the file is CLOSEd by closing both the channels.

To illustrate, Fig. 8-7 is the listing of a subroutine for writing the file documented in Fig. 8-3. Let us go through this subroutine line by line.

Line 7010 calls subroutine 8000, which converts the single record number R to the low-byte and high-byte record numbers RL and RH. This conversion could be done in the file write subroutine itself, but since it is required in all subroutines that write or read random-access files, it is more efficient to keep it separate. (Subroutine 8000 is discussed below.)

Line 7020 OPENs the command channel. Line 7030 OPENs file channel 6. Line 7040 positions the record pointer. Lines 7050-7200 print data to the record with PRINT# statements. Line 7210 CLOSEs the file channel and line 7220 CLOSEs the command channel.

The only difference between the code for writing and reading a file is that PRINT# and INPUT# statements are substituted for one another. Figure 8-8 is the listing of the subroutine for reading the random-access file just described. Compare it with the listing shown in Fig. 8-7.

Now let us consider how to compute RL and RH from R. Figure 8-9 is the listing of a subroutine that converts R to its equivalent low-byte and high-byte form in the variables RL and RH. This subroutine performs two tests and sets RL and RH accordingly. The logic is straightforward; I leave it to your common sense to figure out.

That covers the fundamentals of planning, creating, writing, and reading standard random-access files. If you want records longer than 254 characters, these will not quite do the job for you. The next section shows a little trick to get around this limitation.

Pseudorandom-Access Files

You can make a set of sequential files act as if

```
7250 REM--RANDOM FILE (READ)--
7260 GOSUB 8000:REM CALCULATE RL & RH
7270 OPEN 15,8,15
7280 OPEN 6,8,6,"RANDFILE"
7290 PRINT#15,"P"CHR$(6)CHR$(RL)CHR$(RH)
7300 REM-STRING ARRAY-
7310 FOR A=0 TO 5
7320 INPUT#6,A$(A)
7330 NEXT
7340 REM-REAL ARRAY-
7350 FOR A=0 TO 5
7360 INPUT#6,A(A)
7370 NEXT
7380 REM-INTEGER ARRAY-
7390 FOR A=0 TO 5
7400 INPUT#6,A%(A)
7410 NEXT
7420 REM-VARIABLES-
7430 INPUT#6,A$
7440 INPUT#6,A0
7450 INPUT#6,A%
7460 CLOSE 6
7470 CLOSE 15
7480 RETURN
```

Fig. 8-8. Subroutine to read random-access file.

```
8000 REM--CALCULATE RL & RH--
8010 IF R>=512 THEN RH=2:RL=R-512:GOTO 8040
8020 IF R>=256 THEN RH=1:RL=R-256:GOTO 8040
8030 RH=0:RL=R
8040 RETURN
```

Fig. 8-9. Subroutine to compute Record Low (RL) and Record High (RH) numbers based on single record number R.

it was a random-access file. Since there is no limitation on the length of a sequential file, this permits you to create something that behaves like a random-access file but has records longer than 254 characters.

How do you make a sequential file act like a random-access file? Create a series of files with slightly different names. For example, create a series of sequential files named DATA 1, DATA 2, DATA 3, and so on, for as far on as you need to go. You create these different files by starting with a core file name and then appending a record number to it. The record number then becomes a part of the file name and gives you a way to access the file. For example, create record number 5 in your DATA file, by appending the number 5 to the file name. Later, you can read the file back by appending the record number 5 to the file name and looking for a sequential file with the name DATA 5.

Doing this is not as complicated as it might sound. To illustrate, let us go back to the BASIC sequential file write subroutines discussed earlier and shown in Fig. 8-4. The OPEN statement is contained in line 6020, which looks like this:

```
6020  OPEN 2,8,2,"@0:SEQFILE,SEQ,W":REM
OPEN FILE
```

Now, modify line 6020 by changing it to the following:

```
6020 OPEN 2,8,2,FI$
```

Add the following line earlier in the write subroutine:

```
6015  FI$ = "@0:DATA"+STR$(R)+",SEQ,W"
```

Consider what happens when these two lines execute.

Line 6015 appends the record number, R, to the file name "DATA". FI$ is then set equal to everything on the right of the parenthesis sign. If you look carefully, you realize the FI$ is the right half of what is required in a file OPEN statement. FI$ is then used in line 6020 to complete the OPEN statement.

Now, whatever record number you send to this subroutine is concatenated by line 6015 with the file name, and then line 6020 OPENs the named file. If you send the subroutine R=0, then a file named "DATA 0" is OPENed. If you send it R=23, then file "DATA 23" is OPENed.

Reading the file is just as easy. The same two lines are required, although you must change the W at the end of line 6015 to an R.

Figure 8-10 is the listing of both write and read subroutines for a pseudorandom-access file based on SEQFILE—the sequential file whose write and read subroutines are shown in Figs. 8-4 and 8-5, respectively. The write subroutine in Fig. 8-10 starts at line 7500, and the read subroutine at line 7750. Lines 7510 and 7520 of the file write subroutine define the file name and OPEN the file in the manner just described. The equivalent lines in the file read subroutine are 7760 and 7770, and that is all there is to it.

Pseudorandom-access files are not a substitute for standard random-access files, but they are invaluable when you want a file with longer records than the standard version allows.

Since a pseudorandom-access file is actually a set of sequential files, each record can be created by simply writing to it. A separate act of file creation, as is needed with standard random access files, is

unnecessary. Still, it is smart to create the entire file and write something to every record early as a safeguard against an attempt to read a record that does not exist. To create the file, set the strings and string arrays to something other than the null string, and then set a FOR-NEXT loop in motion and write the file at each record number. (This is the technique recommended for creating standard random-access files, and was described earlier in this section.)

READING THE ERROR CHANNEL

The read and write subroutines described so far lack any error-trapping. The error channel permits you to do a certain amount of this. It has authority with DOS, and while it is OPEN, your

```
7500 REM--PSEUDO-RANDOM FILE (WRITE)--
7510 FI$="@0:DATA"+STR$(R)+",SEQ,W"
7520 OPEN 2,8,2,FI$:REM OPEN FILE
7530 REM-STRING ARRAY-
7540 FOR A=0 TO 5
7550 PRINT#2,A$(A)
7560 NEXT
7570 REM-REAL ARRAY-
7580 FOR A=0 TO 5
7590 PRINT#2,A(A)
7600 NEXT
7610 REM-INTEGER ARRAY-
7620 FOR A=0 TO 5
7630 PRINT#2,A%(A)
7640 NEXT
7650 REM-VARIABLES-
7660 PRINT#2,A$
7670 PRINT#2,A0
7680 PRINT#2,A%
7690 CLOSE 2:REM CLOSE FILE
7700 RETURN
7750 REM--PSEUDO-RANDOM FILE (READ)--
7760 FI$="@0:DATA"+STR$(R)+",SEQ,R"
7770 OPEN 2,8,2,FI$:REM OPEN FILE
7780 REM-STRING ARRAY-
7790 FOR A=0 TO 5
7800 INPUT#2,A$(A)
7810 NEXT
7820 REM-REAL ARRAY-
7830 FOR A=0 TO 5
7840 INPUT#2,A(A)
7850 NEXT
7860 REM-INTEGER ARRAY-
7870 FOR A=0 TO 5
7880 INPUT#2,A%(A)
7890 NEXT
7900 REM-VARIABLES-
7910 INPUT#2,A$
7920 INPUT#2,A0
7930 INPUT#2,A%
7940 CLOSE 2:REM CLOSE FILE
7950 RETURN
```

Fig. 8-10. Subroutines to write (lines 7500-7700) and read (lines 7750-7950) pseudorandom-access file.

C-64 is relatively immune to a crash. It is sort of like the policeman at the scene of a traffic accident—all of those involved stay put, keep their mouths shut, and wait to see what happens next. The policeman takes notes and decides who was at fault. If you ask him, he will tell you, but he does not volunteer anything. While he is there, however, no fights break out, and everything remains under control.

In more literal terms, you can use the error channel to identify and handle disk-related error conditions—mostly the result of human error—that may occur during your program.

For example, if your program requires disk swapping, there is a good chance that at some point the wrong disk will be in the drive, and that your program will attempt to read a file that is not present. You can trap this type of error via the error channel.

The standard and recommended practice in C-64 programming is to have your program OPEN and read the error channel each time it performs a disk operation—that is, attempts to write or read a file. Reading the error channel is, of course, optional. The subroutines presented earlier in this chapter do not do so. I left this code out for simplicity, but in the present section I show how to add it to file write and read subroutines.

A key reason to read the error channel is to identify the type of error condition that occurred. If you have ever written code that produced a disk error (and what C-64 programmer has not?), then you know how frustrating it is to watch the light on your disk drive flash on and off to tell you that something went wrong—without telling you what. This is irritating to programmers, but even worse when it happens to a user. Have you ever had a user tell you that something went wrong during a disk operation without being able to pinpoint the exact error? It is much better if the user can tell you that, say, error 52 occurred, since this defines the error specifically.

The best way to build an error-handling routine is as (guess what) a subroutine. Such a routine must OPEN the error channel, read it, decide whether or not an error condition exists and, if so, display the error information to the user. It requires several lines of code, and it is not efficient to reproduce all these lines in every write or read subroutine.

The code for such a subroutine can be written in a number of different ways. In the following method, the error-handling subroutine is used together with a file write or read subroutine—these two bear a sort of symbiotic relationship. As their interaction can be a bit confusing, let us start with a flowchart that shows what goes on in each subroutine and how the two subroutines interact.

Figure 8-11 is a flowchart showing the functions performed in a write or read subroutine (left), the error-handling subroutine (right), and the interaction between the two subroutines. Each of the steps in these subroutines is numbered. Let us go through them one by one, starting on the left at Step 1.

Step 1 is to OPEN the error channel. The error channel is channel 15, which is the command channel. Note that it is not necessary to OPEN the command channel to write or read a sequential file (see Figs. 8-4 and 8-5), although this channel must be open when working with a random-access file. In other words, code must be added to a sequential-file-handling subroutine to OPEN (and CLOSE) channel 15, although this is not necessary with a random-access file.

Step 2 is to OPEN the file that is to be written or read. Step 3 is to call the error-handling subroutine.

In Step 4, the error-handling subroutine reads in error data over channel 15 with the INPUT# statement. Up to four items of error data are available: Error Number, Error Name, Track, and Block. They are read in with a statement such as INPUT# 15, E1$, E2$, E3$, E4$. If there is no error condition, the VALue of E1$ is zero. Otherwise, it is one of the DOS error numbers. E2$ is the corresponding error message. The third and fourth error terms (E3$, E4$) are of interest when working with Type 1 random-access files (as defined earlier in this chapter), but otherwise they can be ignored.

Step 5 tests E1$ to see whether an error con-

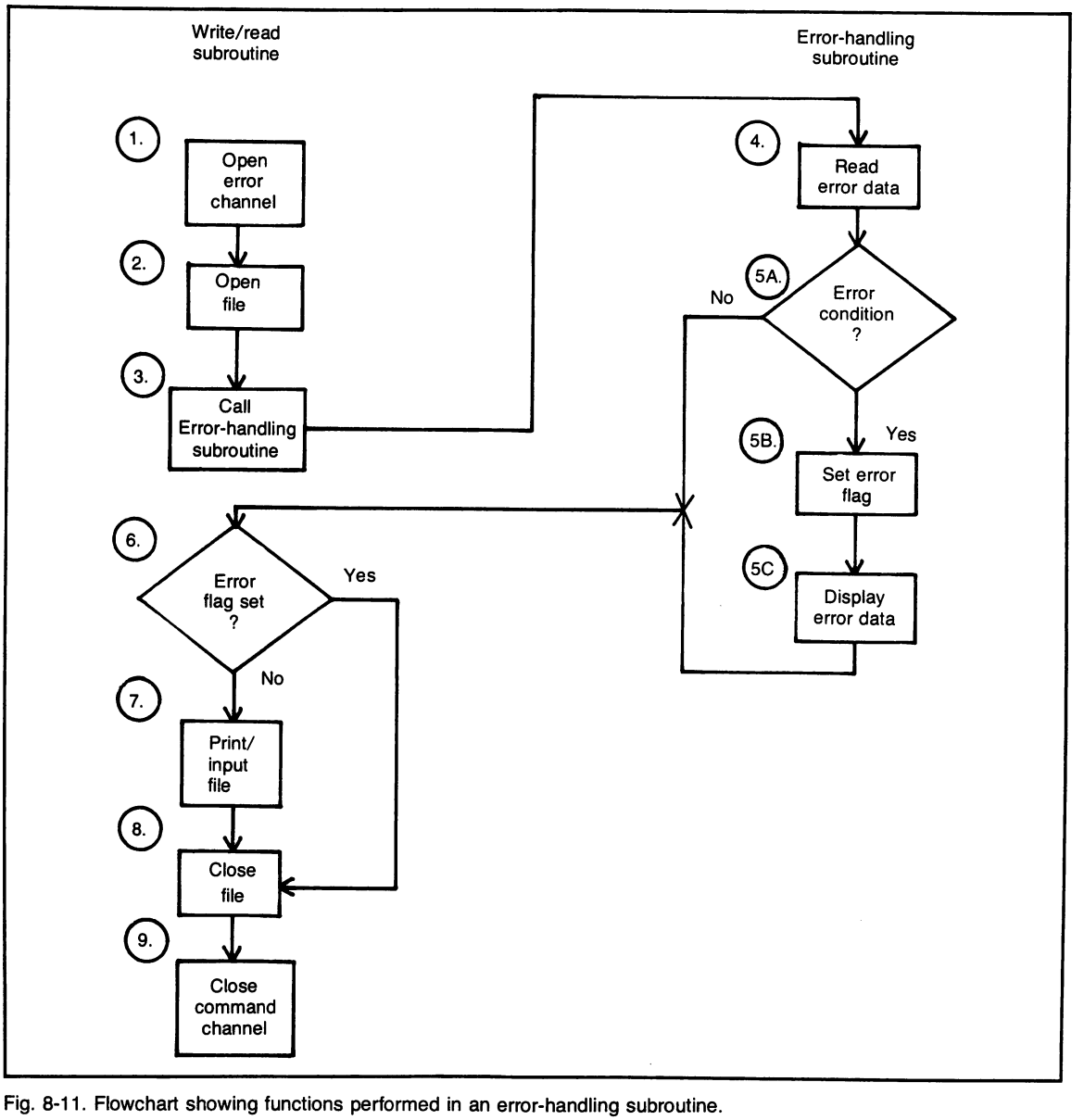


Fig. 8-11. Flowchart showing functions performed in an error-handling subroutine.

dition has occurred; that is, VAL(E1$"≠0). If not, it RETURNs control the file write or read subroutine (Step 6).

If an error has occurred, then the error flag is set (define ER%=1) in Step 5B, error data are displayed in Step 5C, and then control is RETURNed to the file write or read subroutine in Step 6.

At Step 6, an IF-THEN statement tests whether the error flag is set. If not, control flows to Step 6, which PRINT#'s data to or INPUT#'s data from the file. If an error condition was found in Step 6, however, Step 7 is skipped and control flows to Step 8, which CLOSEs the file, and then to Step 9, which CLOSEs the command channel.

Now let us go through the program code, re-

lating it to the logic just described. Figure 8-12 is the listing of a sequential file write subroutine (lines 6010-6090) and its related error-handling subroutine (lines 8100-8210).

The write subroutine creates and writes to a file called "DUMMY". Line 6020 OPENs the command channel so that error data can be read in. (Keep in mind that the command channel does not have to be OPENed to read a sequential file, and that it is included here to support the error-handling subroutine.) Line 6030 OPENs the file. Line 6040 then calls the error-handling subroutine with a GOSUB 8100.

Line 8110 sets the error flag, ER%, to zero. It is initialized at the start of this subroutine since a previous error condition may have caused ER% to be set to 1—which can have consequences later. Line 8120 INPUT#'s two strings from the error channel: E1$ (Error Number), and E2$ (Error Name).

Line 8130 tests whether the VALue of E1$ is zero. If so, then no error condition has occurred, and control is sent to line 8210, which ends the subroutine with a RETURN to the file write subroutine. If the VALue of E1$ is other than zero, then an error has occurred, and lines 8140-8200 are

```
10 REM ************************************
20 REM * NOTE: THIS SUBROUTINE REQUIRES  *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE    *
40 REM * LOADER (FIGURE 4-6)             *
50 REM ************************************
4010 REM--TEMPORARY PAUSE--
4020 SYS C0,0,24,6
4030 PRINT CHR$(18);:REM REVERSE VIDEO
4040 PRINT "[PRESS SPACE BAR TO CONTINUE]";
4050 GET A$
4060 IF A$<>" " GOTO 4050
4070 PRINT
4080 RETURN
6010 REM--WRITE DUMMY FILE--
6020 OPEN 15,8,15:REM OPEN COMMAND CHANNEL
6030 OPEN 3,8,3,"@0:DUMMY,SEQ,W":REM OPEN FILE
6040 GOSUB 8100:REM ERROR TEST
6050 IF ER%=1 GOTO 6070:REM ERROR OCCURRED--EXIT SUBROUTINE
6060 PRINT#3,"COMMODORE 64"
6070 CLOSE 3:REM CLOSE FILE
6080 CLOSE 15:REM CLOSE COMMAND CHANNEL
6090 RETURN
8100 REM--ERROR HANDLER--
8110 ER%=0:REM ZERO ERROR FLAG
8120 INPUT#15,E1$,E2$
8130 IF VAL(E1$)=0 GOTO 8210:REM NO ERROR
8140 REM-ERROR CONDITION-
8150 ER%=1:REM SET ERROR FLAG
8160 REM-DISPLAY ERROR INFORMATION-
8170 PRINT CHR$(147)
8180 PRINT"ERROR: ";E1$
8190 PRINT"TYPE  : ";E2$
8200 GOSUB 4010:REM TEMPORARY PAUSE
8210 RETURN
```

Fig. 8-12. Code showing the relationship between sequential file write subroutine (lines 6010-6090) and related error-handling subroutine (lines 8100-8210).

executed. In this case, line 8150 sets the error flag E%=1, and lines 8170-8190 display the error data. Line 4010 causes a temporary pause until the space bar is pressed, giving the user time to read the error message.

After the RETURN from the error-handling subroutine, control goes to line 6050 of the file write subroutine. This line tests whether the error flag is set. If so (ER%=1), control is sent first to line 6070, which CLOSEs the file, and then to line 6080, which CLOSEs the command channel. This is done without PRINT#ing data to the file in line 6060. If, however, the error flag has not been set, then the test in line 6050 is failed, and line 6060 is executed, PRINT#ing data to the file before CLOSEing it and the command channel.

Try the error-handling subroutine out for yourself. Type in the code shown in lines 6010-8210. Note that line 8200 uses the temporary pause subroutine at line 4010 which, in turn, requires the availability of the machine language subroutines that were presented in Chapter 4. If you want to simplify things, change line 8200 to any type of temporary pause (GET or INPUT) or even leave it out altogether.

After you have typed these two subroutines in, add the following lines:

```
  990  GOTO 10000
10000  GOSUB 6000
```

Now remove the disk from the drive and RUN the program. This produces an error condition as the drive attempts to write the file without a disk in it. If your drive has the same temperament as mine, you should get:

```
ERROR: 74
TYPE: DRIVE NOT READY
```

Incidentally, if an error 74 (drive not ready) error occurs, all subsequent disk errors will also be reported as error 74, regardless of cause.

To add to the fun, convert the write subroutine to a read subroutine and attempt to read a nonexistent file. This should give you:

```
ERROR: 62
TYPE: FILE NOT FOUND
```

The above is the most elementary sort of error-handling subroutine. It accomplishes the functions necessary to get the job done, but nothing more. You can make it fancier by formatting the error display nicely, providing on-screen directions (such as "write down the error number and name and contact the programmer"), or adding additional wrinkles of your own. Consider the user audience, and develop your error-handling routine accordingly. Anticipate what they might do when an error occurs. If you think that additional on-screen directions are needed, provide them. The main goal, of course, is to minimize the damage, whatever it might be, and to collect the error information in sufficient detail that you or another programmer can fix things.

FILE VERIFICATION

Chapter 7 described how to chain between programs using 2.0 level BASIC. One piece of the chaining puzzle had been left out, however. The missing part is a technique to verify that a program file is present on disk before you attempt to chain to it. If you attempt to chain, and the file is not there, your program will crash. For this reason, it is a good idea to make sure that what you want is there before you make the attempt.

This section shows how to verify your program file, or at least do the next best thing. If you do not intend to chain in any of your programs, or if you always keep the same disk in your disk drive, then you can skip this section. It does not have to do with data files, per se. It simply adapts the data file error-handling technique to another application.

Suppose that you have a disk with two programs on it called "PROGRAM A" and "PROGRAM B". Figure 8-13 shows the control logic to follow in chaining between these two programs.

Step 1 is to identify the name of the next program—the program to chain to. Assume that this is PROGRAM B. Step 2 is to verify PROGRAM B—Check whether it is present on the disk.

Step 3 is to make a decision based on the

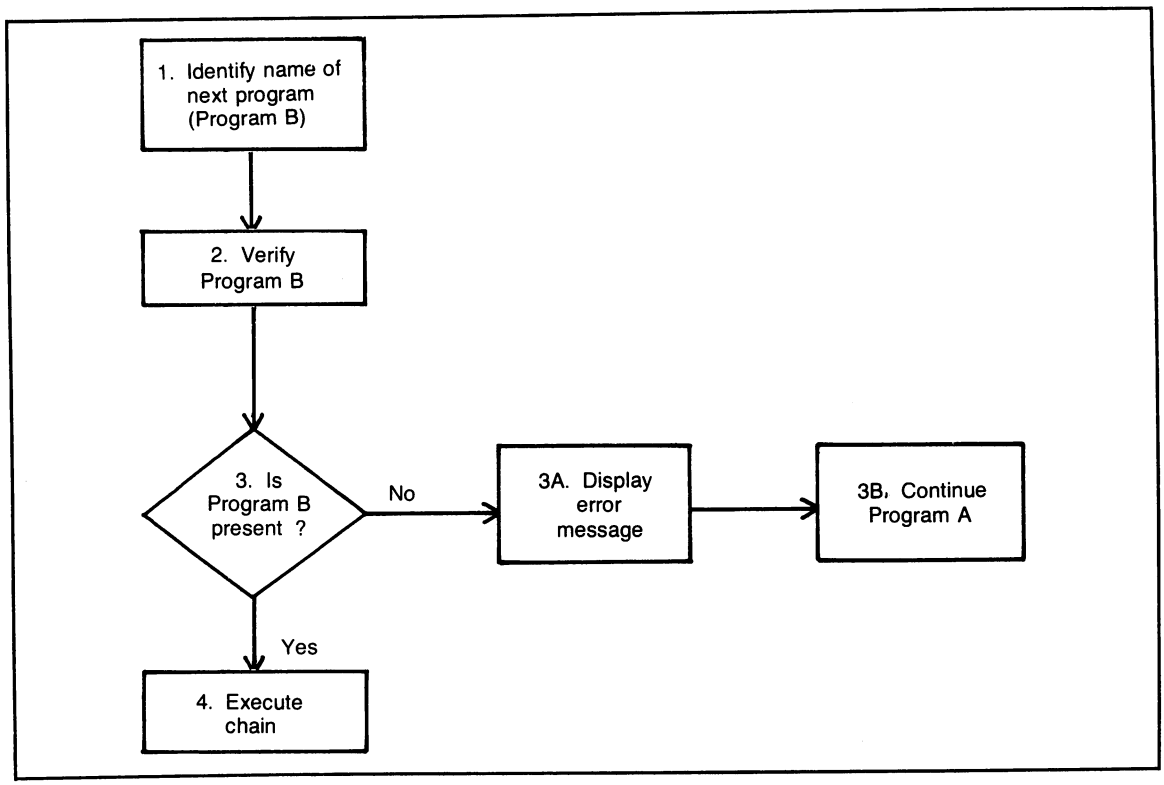


Fig. 8-13. Chain verification logic.

results of Step 2. If PROGRAM B is present, then Step 4 is performed and chaining occurs. If Step 3 indicates that Program B is not present, however, then chaining does not occur. Instead, Steps 3A and 3B are performed: an error message is displayed, and then the current program is continued.

Verification is the safety net. It allows you to back away from chaining if doing so will cause an error condition to occur. The key to this is Step 2, verification. How can you test the disk to see whether or not the program file you want to chain to is present?

The previous section showed how to read the error channel and deal with disk error conditions. Using that technique, you can attempt to OPEN a file and then go to an error-handling subroutine. If the file is not found, then control logic CLOSEs the file and displays an error message. It is possible, but cumbersome and inadvisable, to adapt this technique to work with program files.

Instead, it is better to use a little ploy. The ploy consists of putting a dummy text file on the disk with a name that is logically related to that of the program. For example, since the objective is to chain to a program named PROGRAM B, supposed that you add a text file to the disk named "PROGRAM B.TEXT". Then, when you want to chain to PROGRAM B, you can have PROGRAM A attempt to OPEN the PROGRAM B.TEXT file and use an error-handling routine to see whether it is present. If it is, then PROGRAM B must also be present and it is safe to chain. On the other hand, if PROGRAM B.TEXT is not present, then PROGRAM B is not present, and chaining should not be attempted. The chaining logic is identical to that shown in Fig. 8-13 except that, instead of verifying the next program, you verify its stand-in.

Creating a "stand-in" text file is very simple. Figure 8-14 is the listing of a short program that does it for you. When you RUN this program, it

```
10 REM--CREATE STAND-IN TEXT FILE--
20 A$=""
30 INPUT"PROGRAM FILE NAME: ";A$
40 FI$=A$+".TEXT"+",SEQ,W"
50 OPEN 2,8,2,FI$
60 PRINT#2,"-"
70 CLOSE 2
80 END
```

Fig. 8-14. Program to create stand-in text file for use in program verification.

requests you to enter a PROGRAM FILE NAME. It then creates a short text file whose name consists of the program name with the characters ".TEXT" appended. For example, if you type in PROGRAM B, this program creates a text file named PROGRAM B.TEXT. The content of the file is one hyphen—enough to give the file substance, but not requiring much disk space. The program is very simple, and I leave it to you to figure out.

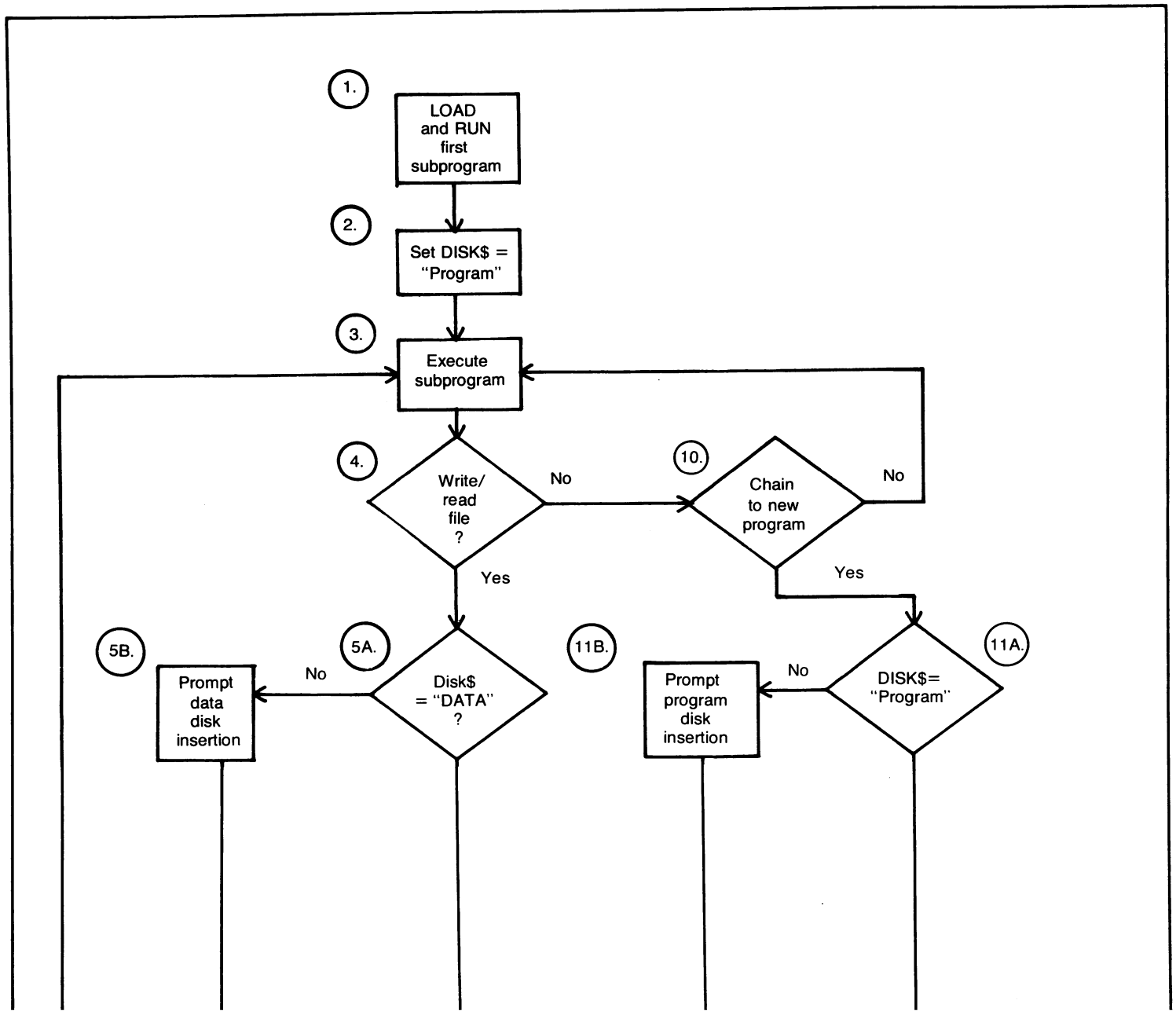
Note that, since file names cannot be longer than 16 characters, if you use long file names you might want to substitute some suffix other than ".TEXT" to program names. If the program name and appended suffix exceed 16 characters, DOS removes the extra characters from the end. This undercuts file verification (see below), since the stand-in file does not have the proper name relationship with the program. The result is that you get an error 62 (file not found) during a chaining attempt.

Now let us examine the code that performs the actual verification. Figure 8-15 is the listing of the verification code. This looks like a subroutine, and is more or less treated like one, but lacks a RETURN statement at the end.

The verification code consists of lines 4800-

```
10 REM ************************************
20 REM * NOTE: THIS SUBROUTINE REQUIRES   *
30 REM * ASSEMBLY-LANGUAGE SUBROUTINE     *
40 REM * LOADER (FIGURE 4-6)              *
50 REM ************************************
4010 REM--TEMPORARY PAUSE--
4020 SYS C0,0,24,6
4030 PRINT CHR$(18);:REM REVERSE VIDEO
4040 PRINT "[PRESS SPACE BAR TO CONTINUE]";
4050 GET A$
4060 IF A$<>" " GOTO 4050
4070 PRINT
4080 RETURN
4800 REM--CHAIN VERIFIER--
4810 FI$=A$+".TEXT"+",SEQ,R"
4820 OPEN 15,8,15:REM OPEN COMMAND CHANNEL
4830 OPEN 2,8,2,FI$:REM ATTEMPT TO OPEN STAND-IN FILE
4840 INPUT#15,E1$,E2$
4850 CLOSE 2
4860 CLOSE 15
4870 IF VAL(E1$)=0 THEN LOAD A$,8:REM IF NO ERROR THEN CHAIN TO NEXT
     PROGRAM
4880 REM-DISPLAY ERROR INFORMATION-
4890 PRINT CHR$(147)
4900 PRINT "ERROR: ";E1$
4910 PRINT "TYPE : ";E2$
4920 GOSUB 4010:REM TEMPORARY PAUSE
4930 GOTO 20000:REM RETURN TO PROGRAM
```

Fig. 8-15. A subroutine (lines 4800-4930) to support program verification before chaining.


1.
LOAD and RUN first subprogram
2.
Set DISK$ = "Program"
3.
Execute subprogram
4.
Write/ read file ?
No
10.
Chain to new program
No
Yes
Yes
5B.
5A.
11B.
11A.
Prompt data disk insertion
No
Disk$ = "DATA" ?
Prompt program disk insertion
No
DISK$= "Program"

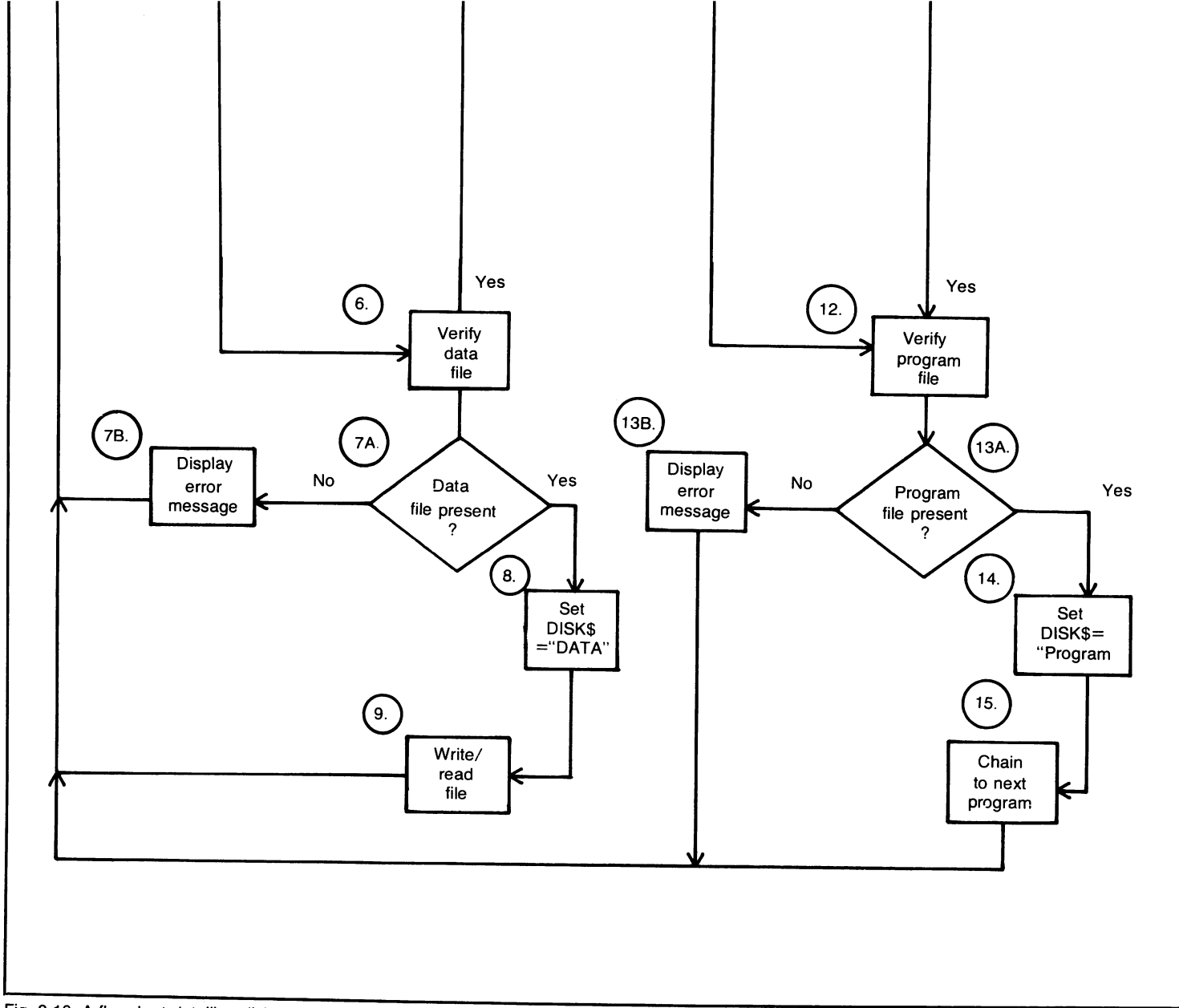


Fig. 8-16. A flowchart detailing disk management logic.

4930, a line range in the Control portion of the subroutines (see Chapter 2). This routine has one argument, A$, which is the name of the program to which to chain. At the point in your program at which it is time to chain, insert lines assigning the program name to the variable A$ and then send control to this routine with a GOTO 4950. For example:

```
20000  A$="PROGRAM B"
20010  GOTO 4950
```

Line 4810 concatenates the program with ".TEXT" and with ",SEQ,R" and assigns the resulting string to FI$ (file name). The string will be used in an attempt to OPEN a text file whose name consists of the program name with the suffix .TEXT appended (the same text file that you created with the program shown in Fig. 8-14).

Line 4820 OPENs the command channel as a preface to error-handling.

Line 4830 attempts to OPEN the text file whose name was constructed in line 4810.

Line 4840 reads the error channel.

Lines 4850 and 4860 CLOSE the file and command channels.

Line 4870 tests E1$ to detect whether an error condition exists. If not (VAL(E1$)=0), then it chains to the next program by LOADing A$. If the test is failed, however, then error information is displayed, the program pauses, and then control is sent with a GOTO back to some strategic point in the program, such as the main menu.

MANAGING DISK-SWAPPING WITH ONE-DRIVE SYSTEMS

Many serious programs require separate program and data disks. Even when there is sufficient room on one disk for both programs and data, many programmers prefer to store programs and data separately. One reason for this is that, whenever a file is written to a disk, there is danger of something going wrong and perhaps destroying a file. Thus, if you prevent users from writing anything to a program disk, you reduce the likelihood of one of your program files being blitzed.

If you have two disk drives, the solution is simple: put the program disk in one drive and the data disk in the other. This is fine for those with two drives, but unfortunately, the majority of C-64 users have but one drive. Consequently, you must make your program smart enough to manage disk swaps; that is, it must keep track of which disk is in the drive and prompt the user to insert the program or data disk as necessary. This section describes how to manage these swaps with a one-drive system. Note that users with two drives must make disk swaps if they have more than one program or data disk. Although I do not cover the two-drive situation, the ideas presented here can be extended to it.

Suppose that you use separate program and data disks and have a one-drive system. How do you manage disk swaps? The blind optimist's approach is to trust the user to know what disk to put in the drive and when. Candide was a blind optimist, if you recall, and his blind optimism did not prove very effective in averting disasters. Analogously, I suggest that you eschew blind optimism and take a more cautious approach to the problem.

Actually, the key to solving this problem—file verification—was presented earlier in this chapter. The section showed you how to verify data files and verify program files (or at least their stand-ins). In other words, you have a method to go beyond blind optimism and to check whether a file, either program or data, is actually present.

The easiest way to explain the management logic is with a flowchart. Figure 8-16 is the disk management logic. (The boxes are numbered to make it easier to talk about them, not because their steps are performed in sequence.) Let us start at the top, and work our way down.

Step 1 is to LOAD and RUN the first subprogram. Step 2 defines the string DISK$="PROGRAM". This string is used to keep track of which disk is in the drive. When a program disk is in the drive, it is set to "PROGRAM". When a data disk is in the drive, it is set to "DATA". It is defined at various points in the program, as described below.

Before setting it to either PROGRAM or DATA, you must be reasonably sure that the ap-

propriate disk is in the drive. It is a safe bet at the beginning of the program that the PROGRAM disk is there, and so it is reasonably safe to set it to "PROGRAM". The user may remove the program disk and immediately insert the data disk but, as you will see, the management logic handles this contingency.

Step 3 is to execute a subprogram. This may be any subprogram in the program. Three things may happen during this subprogram that require a disk swap:

- Write file.
- Read file.
- Chain to new subprogram.

As far as disk management is concerned, writing and reading a file have the same requirements and can be treated identically. Consequently, disk management reduces to two cases: write/read file and chain to new program. These two cases are represented by the two tracks in Fig. 8-16. The first track consists of Steps 4-9. The second track consists of Steps 10-15. Before I get into the content of these two tracks, compare them and note that they are functionally identical. The only difference is that the word DATA is used in one track and PROGRAM in the other. Let us start with the left track.

Step 4 is a decision box that asks whether a file is to be written or read. If not, control goes to Step 10. Assume that the answer to the question posed in Step 4 is yes, that a file is to be written or read.

Step 5A is a decision box that asks whether DISK$="DATA". If it does then presumably the data disk is in the drive. If it does not, then the program disk should be in the drive. If the test is failed, then control goes to Step 5B, which prompts the user to swap disks. That is, the screen might display a message saying something such as:

REMOVE PROGRAM DISK FROM DRIVE
INSERT DATA DISK INTO DRIVE

The program then pauses, giving the user time to make the switch. Control then goes to Step 6. There is no guarantee at this point that the user has made the switch. Moreover, there is no guarantee that the test made in Step 5A is valid. Consequently, like the careful policeman, the program must verify the witness' claims with a fact.

Step 6 is to verify that the data file is present. This is done with an error-handling routine such as that described earlier in this chapter.

Step 7A is a decision box that switches control based on the results of the verification test made during Step 6. If the test is successful, control goes first to Step 8, which sets DISK$="DATA" and then to Step 9, which writes or reads the file. Control then returns to the subprogram. Note that Step 8 is the only reasonable place to redefine DISK$, since it follows verification that the data file is actually present.

Steps 10-15 are functionally identical to Steps 4-9. Step 11A tests for a program disk instead of a data disk. Step 11B displays a message to insert a program disk instead of a data disk. Step 12 verifies a program file instead of a data file, and so on.

How do you do this in actual program code? Not difficult. First, you must define a disk flag. The examples use the string variable DISK$. There is nothing magical about this and you can use anything you like. Some programmers prefer integer variables as flags. It is slightly easier to get an integer variable across the chaining abyss between subprograms, and so you might prefer using something such as DI% as your disk flag, setting it to, say, 0 for a data disk and 1 for a program disk.

Set DISK$ (or whatever flag you use) to indicate a program disk at the start of the first subprogram. Create a subroutine to perform Steps 5A and 5B—it tests whether a data disk is inserted and, if not, prompts the user to insert one. Call this subroutine at the beginning of every file write or read subroutine. It ensures that the user is prompted to make disk insertions before the program attempts to write or read a file.

Create a subroutine to perform Steps 11A and 11B; it tests whether a program disk is inserted and, if not, prompts the user to insert one. Call this subroutine before attempting to chain. It ensures that the user is prompted to insert a program disk when necessary.

At the start of each write or read subroutine, after the file has passed verification by the error-trapping subroutine, insert a line that sets the disk flag to indicate a data disk. (This corresponds to Step 8.)

After veryfing that a program file is present, but before chaining to it, set the disk flag to indicate a program disk. (This corresponds to Step 14.) For example, change line 4870 of Fig. 8-15 to read as follows:

```
4870 IF VAL(E1$)=0 THEN DISK$="PRO
GRAM":DISK$=DISK$+"":LOAD A$,8
```

By following these guidelines, you can be reasonably sure that the right disks are where they should be before performing disk operations. Since users have the ability to take one disk out of a drive and insert another whenever they want to, you never have complete control of the situation. You can, however, be, as I say, "reasonably" sure.

EFFICIENCY AND SAFETY IN WRITING AND READING FILES

When should you write a file? More specifically, at what point in your program code should you call the subroutine that writes a data file? The answer to this question is more elusive than it might seem at first. There are two main factors to consider. The first of these is efficiency, and the second is safety.

Efficiency dictates that you write a file only when necessary. Writing a file involves a time delay. It also produces disk wear. Suppose that your program has a data-entry section that collects several items of data from the user. Your program can write the complete file each time the user enters a data field, but this is not very efficient. It is more efficient to wait until the user enters all fields, and then write.

On the other hand, you have the safety factor. You must write the file often enough so that, in case of a power failure, program interruption, or some other unforeseen event, all of the user's work in entering data does not go to waste.

Obviously, these two factors, efficiency and safety, are somewhat in conflict. You must balance them some way to decide how to handle file writing in your program.

To further complicate the matter, in some programs filing writing should be performed automatically by the program, and in others it should occur only when the user tells the program to write the files. Which way you do this depends mainly upon the long-term value of the data with which the user is working. For example, if the program is used to build some sort of permanent data base, then file writing should usually be done automatically to protect against the user's forgetting to do so. On the other hand, if the program is used to perform some sort of analysis—for example, to evaluate alternative real estate investments—then the user may or may not want to save the data left over at the end. In this case, the file should be written only when the user gives an explicit command to do so.

Making the efficiency-safety trade-off and deciding whether or not to write files automatically require a careful analysis, and I have no simple formula to offer you here. The foregoing is intended mainly to raise your consciousness about these issues and the importance of considering them during design.

While there are no simple formulas for making these design trade-offs, there are some guildelines that can help you write files efficiently and safely, regardless of what tradeoffs you make.

First, never write a file unless it has first been read. If you permit your program to do otherwise, then an existing file may be overwritten. In some cases this may be what the user wants; if so, make sure the user is given the chance to verify the choice of overwriting the file before proceeding.

Second, never write a file unless it needs to be written to. If the file has been OPENed and used in the program, but no change has been made to it, there is no need to rewrite it.

How do you know whether a change has been made? Here, at last, we arrive at the subject of write flags. A write flag is a string or variable that your program sets when a data entry is made. It

follows that if no entry is made, no flag is set. Later on, when it comes time to write the file, your program uses this flag to test whether or not to proceed. For example, suppose that you have this data-entry routine in your program:

```
3400  REM—DATA ENTRY—
3410  FLAG$="UP"
3420  INPUT "TYPE IN YOUR NAME:"; NAME$
```

Line 3410 sets the Write Flag; that is, defines the string FLAG$="UP". This means that the data-entry routine has been executed. The flag is not set otherwise. Now, when your program calls the write subroutine, include a line such as 6410 in it:

```
6400  REM—WRITE FILE—
6410  IF FLAG$< >"UP"THEN RETURN etc.
```

The write subroutine is not executed unless FLAG$ is "UP". (Incidentally, the RETURN at the end of the line 6410 should really be a GOTO to the RETURN statement in the subroutine to conform with good programming practice, but it is a little easier to illustrate this way.) Be sure to reset FLAG$="" after the file has been written.

Now, a bit about safety. If your program writes files automatically, it is a good idea to have it do so before chaining to another subprogram, when exiting a data-entry module, and when the user QUITs the program (for example, by selecting the last option on the main menu). Chaining is always a bit risky, no matter how carefully it is done, and it is best to have those data entries stored in a file when your program starts to walk the tightrope.

If your program has a data-entry module (usually a good idea), then the logical time to write the file is when leaving that module. Presumably, the user enters it to make changes, and leaves it to do something else. Anything can happen afterward, so write the file and get the user off the hook.

If your program takes care of file writing within a data-entry module, then there is no need to worry about signing off with the data still unwritten. Otherwise, this is about the most obvious place to write the file. Be honest. Have you ever QUIT a program after a data-entry session and then realized that you forgot to write the file? If so, remember what it felt like? Enough said.

OPENing and CLOSEing Files

You probably noted that all of the file write and read subroutines presented in this chapter both OPEN and CLOSE the file. DOS does not require you to do this. You can, for example, have your program OPEN a file at the beginning of the program, go on to other business, and not CLOSE it until the end. If this is done, however, there is some danger that an interrupt may disrupt the file since it is not CLOSEd properly.

For this reason, I recommend that you OPEN and CLOSE files according to the pattern illustrated in these subroutines. That is, when your program is to read a file, have it OPEN the file, read it, and then CLOSE it. Later, when it is time to write the file, OPEN it, write it, and then CLOSE it.

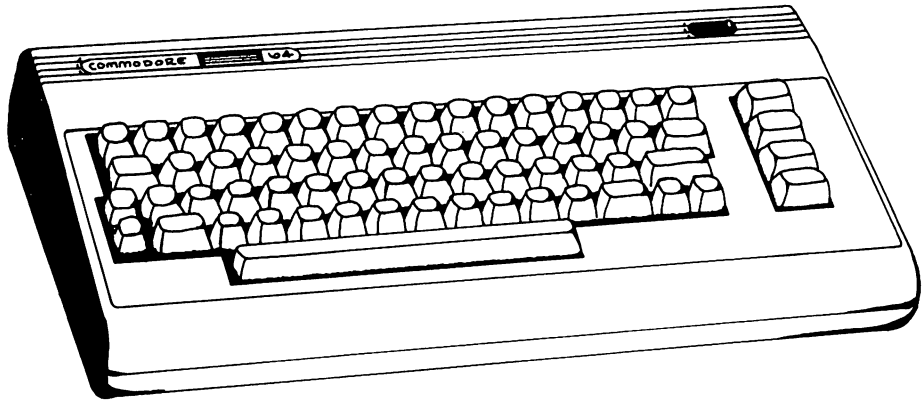
The Well-Read File

Just as you want to be both efficient and safe in the way you handle file writing, you must be concerned with these two factors when it comes to reading files.

As far as efficiency is concerned, do not read a file unless necessary. If your program has several different files and makes use of different ones at different times or in different parts of the program, map out the various requirements and manage them carefully. For example, suppose that your program is organized into three modules. Module 1 requires a file called DATA. Module 2 requires a file called DATA and another called CATEGORIES. Module 3 requires files DATA, CATEGORIES, and BONUS. To assure that you read only the files necessary, insert some file reading control logic to assess which files have been read, check which files need reading, and read the necessary files.

There are a number of other variations on the same theme, but they all boil down to the idea of reading only when necessary.

Chapter 9



Using Your Printer

This chapter covers the fundamentals of using a printer within a program. It does not cover the subject in depth, but stresses a few points of key importance. The chapter is written as if to a reader with a VIC-1525 printer who is generally familiar with its user's manual. If you do not have this manual, you may be at a disadvantage, and it is a good idea to borrow one and, keeping the copyright laws in mind, find a way to make its contents available to you in the future. (Use your imagination.) If you do not have a VIC-1525 printer, do not despair (perhaps you should rejoice, actually). Most of what this chapter covers applies to any type of printer.

The first section covers printer fundamentals—activating and deactivating the printer, simple printing, and enhanced print modes. This section is no substitute for your user's manual, but it hits the highlights in what I hope is a clear and concise fashion.

The second section covers positioning the print head in order to do absolute tabbing—that is, moving the print head where you want it, when you want it there, conveniently. This is an important subject that the user's manual, unfortunately, glosses over.

The final section covers the design of printed reports. It offers a short set of guidelines, most of which derive from the screen design guidelines presented in Chapter 4. The section also presents some subroutines for making report generation easier.

PRINTER FUNDAMENTALS

This section provides a sort of distilled essence of how to use your VIC-1525 printer within a program to print something out. It tells how to activate and deactivate the printer, print data, and use enhanced print modes. The information is presented in directive, cookbook fashion, with a minimum of discussion. If you want more, go to your user's manual. On the other hand, what is presented here should be adequate for controlling your printer in the majority of serious programs.

Activating the Printer

Your program must *activate* the printer, or

wake it up, before it can send data to it. To do this, your program must OPEN a file for the printer and send a command to it. This requires two statements: OPEN and CMD.

The syntax of the OPEN statement is:

OPEN File Number, Device Number

File Number can be any number from 1 to 255. Use 1. Device Number for the VIC-1525 printer is 4. In special circumstances, the Device Number may differ. (See the user's manual for details.) In short, the OPEN statement to wake up your printer is:

```
OPEN 1,4
```

To transfer control to the printer, the program must use a CMD statement. The syntax of this statement is:

CMD File Number

File Number, as noted, is 1. These two lines, then, wake up your printer:

```
OPEN 1, 4
CMD 1
```

Together, they make a neat little subroutine, as shown in Fig. 9-1.

Deactivating the Printer

When your program is through printing data, it must deactivate the printer. If it does not, your computer will remain under the impression that it should send output to the printer, not the screen.

To deactivate the printer, you must first send it a carriage return and then CLOSE the printer channel. The following code does these two things:

```
PRINT#1
CLOSE 1
```

The syntax of the PRINT# statement is as follows:

PRINT# File Number

The argument of the PRINT# statement is the same as the first argument of the OPEN and CLOSE statements. The easiest way to keep things straight is to use a File Number of 1 throughout. Figure 9-2 shows a subroutine that deactivates the printer.

Printing Out Information

After activating the printer, your program directs it to print data with the PRINT# statement. The syntax of this statement is:

PRINT # File Number, Output

Output may be any number or variable, or a character string contained within quotes. Assuming that the file number is 1 (as always), the following are valid PRINT# statements:

```
PRINT#1,X
PRINT#1,A%
PRINT#1,C$
PRINT#1,"CALL ME ISHMAEL"
PRINT#1,55
```

After sending data to the printer, your program must deactivate it, as just described, in order to return output to the screen. The code for ac-

```
8510 REM--ACTIVATE PRINTER--
8520 OPEN 1,4
8530 CMD 1
8540 RETURN
```

Fig. 9-1. A subroutine to activate the printer.

```
8550 REM--DEACTIVATE PRINTER--
8560 PRINT#1
8570 CLOSE 1
8580 RETURN
```

Fig. 9-2. A subroutine to deactivate the printer.

tivating or deactivating a printer is simple, and there is no requirement to use subroutines to perform these two functions. Nonetheless, I recommend that you do so, since it saves you the effort of recalling or looking up the code each time you must add the functions to your program. In addition, if your program activates and deactivates the printer at more than one location in the program, it is more economical to use subroutines.

The following illustrates the use of the subroutines as well as the PRINT# statement to create a hard copy of some of Melville's most memorable words:

```
10000  REM—PRINT PROGRAM—
10010  GOSUB 8500:REM ACTIVATE
PRINTER
10020  PRINT#1, "CALL ME ISHMAEL"
10030  GOSUB  8550:REM DEACTIVATE
PRINTER
```

Type in the two subroutines, and add a line before them with a GOTO 10000 in order to try this little program out.

Enhanced Printing

It is easy to do both reverse and double-width printing with the C-64, or to do both in combination.

Reverse printing is activated with CHR$(18) and deactivated with CHR$(146), just as if you were printing data to the screen. Also, like printing to the screen, reverse mode is deactivated by a carriage return. The CHR$(146) is only necessary to go from reverse to normal mode prior to a carriage return.

The following program prints the quoted information on line 10020 in reverse mode and that on line 10025 in normal mode:

```
10000  REM—PRINT PROGRAM—
10010  GOSUB 8500:REM ACTIVATE
PRINTER
10020  PRINT#1, "CALL ME ISHMAEL"
10025  PRINT#1, "I LIKE NANTUCKET"
10030  GOSUB 8550:REM DEACTIVATE
PRINTER
```

Try it.

By making the following change to line 10025, the first part of the quoted information can be printed in reverse mode, and the second in normal mode:

```
10020  PRINT#1,CHR$(18)"CALL  ME
       ISHMAEL"CHR$(146)"I LIKE
       NANTUCKET"
```

Double-width printing is activated with CHR$(14) and deactivated with CHR$(15). Double-width printing is not deactivated by a carriage return. It must be turned off with CHR$(15) or everything printed comes out double width. The following program prints the quoted information on line 10020 in double width and that on line 10025 in normal width:

```
10000  REM—PRINT PROGRAM—
10010  GOSUB 8500:REM  ACTIVATE
PRINTER
10015  PRINT#1,CHR$(14):REM  TURN
ON DOUBLE-WIDTH MODE
10020  PRINT#1,"CALL ME ISHMAEL"
10021  PRINT#1,CHR$(15):REM  TURN
OFF DOUBLE-WIDTH MODE
10025  PRINT#1,"I LIKE NANTUCKET"
10030  GOSUB 8550:REM DEACTIVATE
PRINTER
```

Double-width printing is handy for titles, and for other printed information that you want to make stand out. Reverse mode is interesting, but it is less readable than normal mode, and its practical utility is questionable. It is probably very useful for something (for example, to stress the word "very" in this sentence for emphasis).

POSITIONING THE PRINT HEAD

In order to generate professional-looking reports, you must be able to control the position of the print head when it starts printing. Some BASICs enable you to position the print head to an absolute tab location with a TAB statement. This is not possible with C-64 BASIC because the printer de-

termines its current position on the print line based on the cursor's position on the screen, not on the printed page. Each time you tab, the print head moves the total distance from the left margin to the specified tab position.

To illustrate, type in this little program and try it. (Make sure that the printer activating and deactivating subroutines shown in Figs. 9-1 and 9-2 are included in the program, and that you include a GOTO statement preceding them to send control to line 10000.)

```
10000  REM—TAB TESTER 1—
10010  GOSUB 8500:REM ACTIVATE
PRINTER
10020  PRINT#1,TAB(10) "ROMEO";
10030  PRINT#1,TAB(20) "JULIET";
10040  GOSUB 8550:REM DEACTIVATE
PRINTER
```

If you try this out, you discover that "ROMEO" is printed at tab position 10, but that "JULIET" is printed 20 spaces to the right, at tab position 30. Now remove the semicolon from line 10020. This produces a carriage return at the end of the line. As a result, "JULIET" is now printed at tab position 20 on the next line. In short, TAB works fine to print one item on a line, but does not position the print head correctly relative to the left margin if you attempt to print more than one item. It always moves the print head as many spaces to the right of the present position as the argument of the TAB statement.

Now change the TABs to SPCs. RUN the program with and without the semicolon at the end of line 10020.

The results? Same as before—SPC and TAB do the same thing. Still, in a certain sense this is an improvement. SPC does what you expect it to—move so many spaces over from the last item printed or from the left margin. TAB does not; it does less than you might expect it to, less than it can do when used for printing information on a video display.

Neither of these statements, TAB or SPC, is very flexible. Each allows you to position precisely and then print just one item. After that, everything is relative. This is fine for printing information whose length you know—such as the headings in a column of figures—but it does not work well for printing information whose length may vary. For example, if you want to print a list of names, you cannot expect them all to be of exactly the same length. The same is true of numbers. In fact, numbers are even more difficult to deal with, since you need to align them on the decimal point, that is, $ format them. Evidently, neither TAB nor SPC quite cuts it.

Fortunately, there is another way to control the location of the print head. Once the printer has been activated, the CHR$(16) code is used to move the print head to a specific tab location, from column 0 through column 79. The syntax of the statement to print something at a particular tab location is as follows:

```
PRINT#1,CHR$(16)"TAB" Output
```

For example, to print "ROMEO" starting at column 10, use this statement:

```
PRINT#1,CHR$(16)"10"+"ROMEO"
```

Type in and try this modified version of the program used earlier to print the names of Shakespeare's most famous hero and heroine:

```
10000  REM—TAB TESTER 2—
10010  GOSUB 8500:REM ACTIVATE
PRINTER
10020  PRINT#1,CHR$(16)"10"+"ROMEO";
10030  PRINT#1,CHR$(16)"20"+"JULIET"
10040  GOSUB 8550:REM DEACTIVATE
PRINTER
```

RUN this program and, lo and behold, it prints "ROMEO" starting at column 10 and "JULIET" starting at column 20.

Remove the semicolon at the end of line 10020 and "JULIET" still prints starting at the same location. At last, absolute tabs!

Is it really that easy? Almost, but not quite.

There is a little glitch in the way this works that is best illustrated by example. Change line 10020 to look like the following and then RUN the program again:

```
10020  PRINT#1,CHR$(16)"5"+"ROMEO";
```

If you try this, you discover that the first character of the name is lopped off and what is left is printed at column 50. The problem is that the TAB value must always consist of two characters, even if it is less than 10. Change the "5" to "05" and try again.

Now it works properly. We seem to be on to something here: however, it is a bit awkward to use the quoted TAB value ("05") format everytime you want to print something. It would be much more convenient to be able to supply an argument such as, say, T (for TAB value), and then have the print head go where you tell it to go. It is a nuisance to worry about learning zeros and that sort of thing.

How about using the STR$ function to convert T to string form? This seems like a possibility, but it has one shortcoming, again best illustrated by example. Try this version of line 10020 in the program:

```
10020  PRINT #1,CHR$(16)STR$(10)
"ROMEO";
```

When you RUN this, it prints "0ROMEO" (as if mimicking Juliet), starting at column 1. The problem here is that the STR$ function adds an extra space to the front end of the number it converts. The printer reads this space as a zero, the next number as the TAB, and assumes that the third number (0) is part of the output. A straight STR$ conversion has its problems, too.

What all of the foregoing illustrates is that absolute tabbing, desirable as it is, is not simple. The solution to the problem is to create a subroutine that takes the arguments T (TAB) and Z$ (output string), uses logic to convert T, and then prints Z$ where it belongs. This subroutine is shown in Fig. 9-3. Let us go through it line by line.

Line 8610 converts T to string form as TAB$.

```
8600 REM--TABULATOR--
8610 TAB$=STR$(T)
8620 L=LEN(TAB$)
8630 IF L=3 THEN TAB$=RIGHT$(TAB$,2)
8640 PRINT#1,CHR$(16)TAB$;Z$;
8650 RETURN
```

Fig. 9-3. A subroutine to perform absolute tabulation on the printer.

The subroutine assumes that the programmer sends it arguments between 1 and 80—either one- or two-digit numbers. As a result, TAB$ consists of either two or three characters.

Line 8620 computes the length, L, of TAB$. Line 8630 tests the length of TAB$ and, if it is 3, removes the leftmost character (an added space). This assures that two-digit TAB values do not have an extra space at the front. This line does not affect one-digit TAB values. Their extra space is not a problem, since it is treated as a leading zero, and is necessary for proper tabbing.

Line 8640 PRINT#s the output string Z$, starting at position TAB$. The line ends with a semicolon so that no carriage return is sent and so that more than one item can be printed on a single row. After the final item has been printed, program code must include a PRINT#1 statement to send the last carriage return. The subroutine ends on line 8650 with a RETURN.

The following listing shows how to use this subroutine to print the two names where you want them:

```
10000  REM—TAB TESTER 3—
10010  GOSUB 8500:REM ACTIVATE
PRINTER
10020  T=10:Z$="ROMEO":GOSUB 8600
10030  T=20:Z$="JULIET":GOSUB 8600
10035  PRINT#1:REM FINAL CARRIAGE
RETURN
10040  GOSUB 8550:REM DEACTIVATE
PRINTER
```

Type in the subroutine shown in Fig. 9-3, and

then adds this little program to demonstrate how the subroutine works to yourself. Change the TAB values, and try some other variations of your own. Subroutine 8600 is extremely important, since it gives you a great deal of added control of your printer.

DESIGNING PRINTED REPORTS

A printed report is the hard-copy output produced by your computer. The report may contain words, numbers, graphics, or a combination of all three. A printed report has two main advantages over a screen display. First, it is permanent. It goes on existing after you turn your computer off—a day, week, month, or a year later you can still hold it in your hand. Second, it can be as long as you want to make it. You are not limited to the narrow window of your video display.

One disadvantage of a printed report is that it exists outside the context of a computer program. That is, when you use a program and work your way to a particular display screen, you view that screen in relation to the screens that came before and the particular path you followed. On the other hand, when you pick up a printed report, that is all there is. It is like pulling a page out of a book without being able to see what came earlier. This is one reason why it is important to identify your reports clearly. Put a title or label on each report, and identification information on the different parts of the report. It often is important to include information that you would not put on a screen report—for example, a date. When you use a computer program, you know the date. When you pull your printed budget report out of the file, however, you may not know when, exactly, the report was generated.

The general lesson here is to take into account the effects of time on the interpretation of the report. What will the report user need to know? It may be the report generation date, who generated the report, certain report generation factors, or whatever. You must decide.

General Design Guidelines

Most of the guidelines for screen design covered in Chapter 4 apply also to the design of printed reports. Since you are already familiar with them from Chapter 4, I will just mention them here, rather than describing them in detail. These rules are:

- Title the report and its various sections.
- Divide the report up into logical areas.
- Presenting text: Use uppercase and lowercase; avoid abbreviations and jargon; use a simple, direct style, with commonplace words; break extended text up into short paragraphs; present procedures in list form.
- Presenting numbers: In displaying columns of numeric information, separate columns by at least two blank spaces; align columns on the decimal point.

Plan your report carefully with a report design matrix (Fig. 4-16) before you start coding.

Title Center and Print Subroutine

Chapter 4 presented a subroutine for centering and printing the character string T$ (title) on the video display (see Fig. 4-17). With minor modifications, this subroutine can be adapted to center and print titles on an 80-column hard-copy report.

Figure 9-4 is the listing of a subroutine to center and print a title on a report. The subroutine has one argument, T$, —the title that is to be centered and printed. This subroutine requires that the printer be active; that is, that subroutine 8500 (or equivalent code) has been executed prior to calling it. It then computes the appropriate tab location and prints the title on the report. The printer must be deactivated afterward.

```
8700 REM--CENTER & PRINT T$--
8710 T=(79-LEN(T$))/2
8720 PRINT#1,TAB(T) T$
8730 RETURN
```

Fig. 9-4. A subroutine to center and print a title on an 80-column printer.

To illustrate its use, suppose that you want to print the title "POWDERMILK BISCUITS" on a report. The following code does it:

```
10000  REM—PRINT TITLE—
10010  GOSUB 8500:REM ACTIVATE
PRINTER
10020  T$="POWDERMILK BISCUITS"
10030  GOSUB 8700:REM CENTER AND
PRINT TITLE
10040  GOSUB 8550:REM DEACTIVATE
PRINTER
```

Note that this subroutine does not work properly if you print the title in double-width mode, since the printer then computes everything as if the display had only 40 columns. In order to center and print double-width titles, you must modify line 8710 as follows:

```
8710 T=(39-LEN(T$) )/2:REM COMPUTE TAB
FOR DOUBLE-WIDTH TITLE
```

$ Formatter

Chapter 4 also presented a subroutine for $ Formatting (Fig. 4-30), which can be easily adapted to format numerical outputs on your printer. Figure 9-5 is a modified version of this subroutine which enables hard-copy printout. Because this subroutine was covered in Chapter 4, I will forego an explanation of it here and focus only on the one change necessary to make it work with a printer.

The change made to this subroutine is the addition of line 2075. This line tests for P%=1, and, if the test is passed, the rest of the line is executed, culminating in a call to subroutine 8600 (the tabulator subroutine discussed earlier in this chapter). P% is the printer flag. It has two possible values: 1 (printer active) and 0 (printer inactive). When you want to use this subroutine to print something on your printer, set it to 1. When you are through, set it back to 0. The easiest way to take care of setting and unsetting the printer flag is to add lines to the Printer Activation subroutine (Fig. 9-1) and Printer Deactivation subroutine (Fig. 9-2).

If P%=1, then line 2075 calculates the T (TAB) argument for the tabulation subroutine based on the calculations made in lines 2010-2070 of the $ Formatter subroutine. You must, however, provide two other arguments: N (number to format) and N0 (number of decimal places). Again, this subroutine does not activate or deactivate the printer, and you must do this separately in your own program code.

The following listing illustrates the use of this subroutine within an actual program:

```
10000  REM—$ FORMATTER
DEMONSTRATION—
10010  GOSUB 8500:REM ACTIVATE
PRINTER
10020  P%=1:REM SET PRINTER FLAG
10030  N=1.234567:REM DEFINE
NUMBER TO FORMAT
10040  T=25:REM SET TAB
10050  N0=3:REM SET NUMBER OF
DECIMAL PLACES
```

```
2000 REM--$ FORMATTER--
2010 N=INT(N*10↑N0+.5)/10↑N0:REM SET DECIMAL PLACES
2020 N$=STR$(N)
2030 L=LEN(N$)
2040 T0=L+1:REM INITIALIZE TAB OFFSET
2050 FOR A=1 TO L
2060 IF MID$(N$,A,1)="."THEN T0=A:REM LOCATE DECIMAL POINT
2070 NEXT
2075 IF P%=1 THEN T=T-T0:Z$=N$:GOSUB 8600:GOTO 2090:REM HARD COPY
2080 PRINT TAB(T-T0)N$;:REM VIDEO DISPLAY
2090 RETURN
```

Fig. 9-5. $ Formatter for use in formatting printed reports.

```
10060  GOSUB 2000:REM CALL $ FOR-
MATTER SUBROUTINE
10070  PRINT#1
10080  P%=0:REM UNSET PRINTER
FLAG
10090  GOSUB 8550:REM DEACTIVATE
PRINTER
```

The REMarks make this code fairly self-explanatory, and so I leave it to you to sort it out.

I suggest that you add this line to the Printer Activation subroutine (Fig. 9-1):

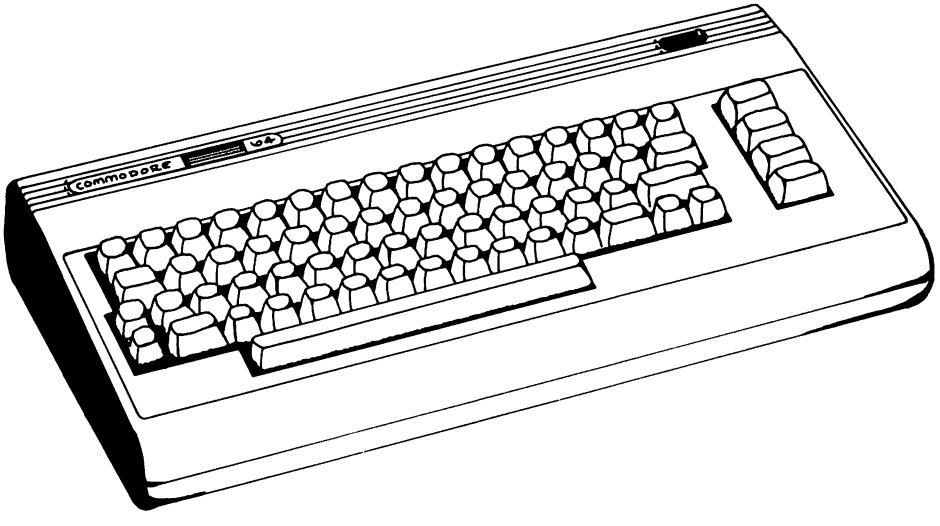
```
8515  P%=1:REM SET PRINTER FLAG
```

Add this line to the Printer Deactivation subroutine (Fig. 9-2):

```
8555  P%=0:REM UNSET PRINTER FLAG
```

Adding these lines saves the requirement of setting and unsetting the flags in program code.

Chapter 10



User Documentation

User documentation is the information that tells people how to use your program. Usually it is in written form and is labeled a "user's guide" or something similar. You can, however, also have documentation within your program. A help screen is one example of such documentation.

In the last few years, people involved in the microcomputer industry have been paying increased attention to user documentation. Program publishers, particularly, have come to realize that the quality of a program's documentation may very well decide its fate. If its documentation is good, then program purchasers will be able to learn about the program quickly and use it efficiently. On the other hand, if the documentation is poor, then they must stumble along on their own and learn by trial and error. When they are forced to do this, often they feel frustrated, angry, and cheated enough to return their program to the computer store, issuing a vote of no confidence in the publisher. Does this sound familiar? You may have done this yourself; I have.

On the other hand, if you write programs strictly for yourself, you probably do not really need a user's guide. You have what is necessary in your head. User's guides are important when you write programs that others use, or that you have ambitions of publishing.

The logical person to write a program user's guide is the program's author. This does not always work out well, however. First, many programmers do not like to write documentation, or feel that they are not very good at it. A few hard-headed programmers do not actually see the need for it. Because they understand how their program works so thoroughly, it may be difficult for them to take the point of view of a program user who is completely naive about it. As a consequence, the program may be poorly documented, if at all.

On the positive side, the user documentation being prepared for commercial microcomputer programs is definitely improving. There is also a good deal more awareness these days about the importance of such documentation. This awareness

is having its effect on the consciousness of programmers—more and more they see the point.

Writing a user's guide or designing help screens for your program is no party. The subject itself does not cause one's pulse to quicken with excitement, either. Still, it is a very important subject for anyone who writes computer programs.

It is important, for example, to the novice programmer, Fred, whom we introduced at the end of Chapter 3. You may recall that Fred, in a quest for information concerning system documentation, sought out and visited The High Guru of Programming, from whom he learned a version of The Sublime Truth (or something like that) concerning system documentation.

Since our current subject is of similar ilk, let us rejoin Fred as he once again confronts The Master. You will recall that Fred has trekked many miles and undergone many trials to reach The High Guru. The scene is as before: Fred sits on the floor of the great hall and the master—an elderly man with a long, gray beard, dressed in saffron robes—sits in the lotus position atop a high, golden pedestal.

Once again, the questioning begins:

Tell me, master, what is user documentation?

User documentation explains your program to users. It comes in two forms: external and internal. External documentation is on paper. Internal documentation is within the program. It consists of help screens, directions, and other information that helps the user with the program.

Why should I provide user documentation?

Idiot! If you do not, how is the user to know what to do with your program? People do not pick up such information by ESP, you know.

Besides, preparing good user documentation is a humane, social act. It is an attempt to communicate information to others and to spare them the frustration, pain, and anger of attempting to use a program without fully understanding it. Do you grasp these concepts?

Yes, master. I humbly beg your forgiveness for my stupidity.

The master forgives all.

You mentioned two types of documentation, master—external and internal. Which is best?

It is not an either-or question. Both have advantages and disadvantages. Look at them as complementary.

Internal documentation is good for presenting reminders, warnings, quick reference information, and other things that the user might want to know on the spot without having to look in a manual. It is not very good for presenting long and detailed descriptions of how a program works, or other information that requires extended text.

Your external documentation, or user's guide, should be the bible for your program. Put the details in it.

Should some information go in one and not the other?

There is no simple answer to this question. It depends on your program. In general, however, the best practice is to make sure that your external documentation contains everything, even what is in the internal documentation. Another way of looking at it is that the internal documentation contains highlights of the external documentation.

Where should I start in preparing user documentation?

The same place you start in designing your program. At least I hope you do. Do you follow?

With the user, master?

Good! I have hope for you. Decide who your user will be. Try to determine as much about the user as you can—age, intelligence, education, computer sophistication, and so forth. It is impossible to know everything, of course, but the type of program you are writing gives you a good idea. For example the guide for your game program should be written as if to a child, since many users

will be children. The guide for your real estate investment analysis program should be written to an adult with financial sophistication, but not great computer sophistication. Your assembly-language programming utility should be written as if to a sophisticated programmer. Know thy audience, and write accordingly.

What should I put in my written user's guide?

This depends upon the type of program. The more complex the program, the more you should include.

Every user's guide needs a section that tells how to set up the program. It explains what hardware is required to use the program, any special software requirements, and how to get the program up and running.

Many user's guides contain a tutorial. A tutorial is a step-by-step lesson that leads the user by the hand through the program. It tells the user to do certain things at the computer, and thereby demonstrates the program's features. It also helps the user build confidence by using the program successfully. In complex programs such as word processors, data base managers, and such, a tutorial is required. Simpler programs do not require it as much, but it is always a good idea to include a tutorial anyway.

Every user's guide should contain reference information that explains how to use the program. It differs from a tutorial in that it is comprehensive and does not actually demonstrate every detail of the program with the user participating. It contains the same basic source information, however—in greater detail, of course—and from it the user can figure out how to work the program. The reference information should cover such things as the required procedures for executing the program's functions, descriptions of commands, and all the other technical details required for using the program. One way of thinking of it is as containing the program's procedures and vocabulary. The procedures are the steps the program user must take to make the program do its job. The vocabulary is the content of commands, special terminology used in the program, and the names of everything.

Just providing this information is not enough, however. You must also make it easy for the user to find and make effective use of what is there. Make sure you include a table of contents and an index. They are extra work for you, but they make it much easier for the user.

In writing your guide, remember the value of concrete examples. You cannot simply explain how to perform some program function abstractly. You must make it real to the reader by using specific examples that illustrate your concepts.

Tell me about internal documentation, master.

First, make it voluntary, not required, for the user to review such information. A help screen helps the user the first time he or she sees it, but not the fourth time, and certainly not the 50th. Let the user decide when to look at help information.

Make appropriate help information available where and when the user needs it. Do not put all your help information in one part of the program where it must be accessed at one time. Rather, distribute it throughout the program so that the user can get the answer to the question being asked while using the program. For example, if the user is performing some complex data-entry function, and gets confused, make it possible to access a help screen in that part of the program.

What kinds of information should be included in internal documentation?

The most obvious things to put there are simple things such as warnings, directions for performing a procedure, a brief explanation of how the particular part of the program works, or lists and definitions of relevant program vocabulary (such as names of commands, and programs). You might, for example, make it possible for the user to call up a help screen with such information by pressing one of the function keys. Things such as this might be regarded as reference information.

You can also be more ambitious. You might, for example, create an animated demonstration that

shows on the screen how the program works. You know, have the program type in the entries, generate the screens, and the like—and every once in a while display a screen with a written explanation of what is happening. A demonstration such as this leaves the user more or less passive, watching what is happening, but not really participating in the action.

What is much nicer is to have an interactive tutorial that is a sort of animated extension of the written tutorial in the user's guide. You might create a set of screens that enables the user to try out various aspects of the program, all the while providing feedback on the screen themselves or with separate screens that interpret the user's entries and tell what to do next.

Thank you, master. I am ready to begin, now. Do you have any final advice for me?

The user. Simply the user. Think of the user before you start writing, and as you write your documentation.

Also think of the user after you have finished the job. If you can, try out your attempts on users and listen to what they say. Modify your documentation based on their comments.

Remember, your documentation is for users. It is not enough for you or your sophisticated programmer friends to like what you have done. Your users must like it, too.

I have listened, learned, and will abide by what you say, master. I have another question.

Ask it.

How did you become so wise?

I was born that way. Any other questions?

Yes, master. You do not have any electricity up here in the Ashram. How do you run your computers?

We do not have computers. We had one once, but it distracted us from our contemplations, and so we painted it gold and made it into the pedestal on which I sit.

But then, master, how do you know so much about computers?

Recognize, novice, that in life some things are simply unexplainable.

Index

Serious Programming for the Commodore 64

by Henry Simpson

- Practical techniques for overcoming machine limitations and increasing the programmer's control of the computer!
- Details on how to design, test, and evaluate programs for clarity, accuracy, and ease of use!

Here's your guide to systematic program design that shows how to avoid usual trial and error methods. Packed with tested techniques and expert advice, it gives practical ways to overcome limitations imposed by the C-64's documentation, 2.0 BASIC, DOS system, and printer commands. Using the author's module system, you can successfully tackle a far wider range of applications programs than initially appear possible on the Commodore 64.

Simpson provides hands-on guidance in developing system documentation so programs will be easier to interpret, debug, or modify. Topics covered include: selection of hardware and software configuration, how to design display screens, how to develop crash-proof routines for collecting data inputs, and how to master program control techniques like menus, simple choices, and typed-in commands.

Using sample modules, the author shows how to link programs together using a chaining technique, plus he gives foolproof methods for handling C-64 disk files and building subroutines to simplify and enhance printer use.

Designed for advanced programmers, this guide is an essential tool for transforming the C-64 into a serious and productive business machine.

The author, Henry Simpson, is a senior scientist at Anacapa Sciences, Inc., Santa Barbara, California, and previous West Coast Editor of Digital Design magazine. He has developed programs for a variety of microcomputers and his articles have appeared in many computer magazines.

TAB TAB BOOKS Inc.
Blue Ridge Summit, Pa. 17214

FPT >> 9.95

ISBN 0-8306-1821-X

PRICES HIGHER IN CANADA

965-1284