Advanced BASIC Programming for the Commodore 64 and Other Commodore Computers

Michael Richter

Advanced BASIC Programming for the Commodore 64 and Other Commodore Computers

Editor-in-Chief: David Culverwell
Executive Editor: Terrell Anderson
Production Editor: Sandra Tamburrino
Text Design: Paula Huber
Cover Design: Don Sellers

Indexer: Leah Kramer
Typesetter: Action Comp Co., Inc., Baltimore, MD
Printer: R. R. Donnelley & Sons Co., Harrisonburg, VA
Typefaces: Trump Mediaeval (text), Antique Olive (display), OCR-A (programs)

Advanced BASIC Programming for the Commodore 64

and Other Commodore Computers

Michael Richter

Robert J. Brady Co., Bowie, MD 20715
A Prentice-Hall Publishing
and Communications Company

Advanced BASIC Programming for the Commodore 64 and Other Commodore Computers

Library of Congress Cataloging in Publication Data

Richter, Michael, 1939–
Advanced BASIC programming for the Commodore 64 and other Commodore computers.

Includes index.
1. Commodore 64 (Computer)—Programming.
2. Commodore computers—Programming. 3. Basic (Computer program language) I. Title.
QA76.8.C64R52 1983 001.64'2 83-15604

ISBN 0-89303-302-2

Prentice-Hall International, Inc., London
Prentice-Hall Canada, Inc., Scarborough, Ontario
Prentice-Hall of Australia, Pty., Ltd., Sydney
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Prentice-Hall of Southeast Asia Pte. Ltd., Singapore
Whitehall Books, Limited, Petone, New Zealand
Editora Prentice-Hall Do Brasil LTDA., Rio de Janeiro

Printed in the United States of America

84 85 86 87 88 89 90 91 92 93 10 9 8 7 6 5 4 3 2 1

CONTENTS

1 Introduction **1**
1.1 Prerequisites and purposes 2
1.2 Principles of BASIC 4
1.3 Interoperability 5

2 Writing for the User **9**
2.1 Program planning 11
2.2 Program organization 14
2.3 Structured programming 17
2.4 Program documentation 19
2.5 Testing 21

3 Mechanics of a Program **25**
3.1 Line numbers 26
3.2 Commands 28
3.3 Variables 29
3.4 Arrays 31
3.5 Bits, bytes, characters, and numbers 33
3.6 Keyboard input 35
3.7 FOR . . . NEXT loops 38
3.8 Machine-language interface 40
3.9 Odds and endings 41
3.10 Coding tricks—DEF and ON 43

4 Devices **47**
4.1 Un-conventions 49
4.2 Printers 51
4.3 Disks and drives 53
4.4 Disk files 55
4.5 Disk commands 57
4.6 Tokens 59
4.7 Using files 61
4.8 Sorts and searches 63

5 SUPERLIST—an Example & a Tool **67**
5.1 SUPERLIST initialization 68
5.2 SUPERLIST utilities 70
5.3 Starting a SUPERLIST 71
5.4 Processing a line 72

5.5 Printing the references 74
5.6 What doesn't work 75
5.7 SUPERLISTing 76
5.8 SYS CHECKSUM 81
5.9 TURTLEWALK 85
5.10 DOMINOES 90

6 Pursuing a Project **97**
6.1 The first call 98
6.2 First meeting 100
6.3 Structuring the system 102
6.4 Inventory control (customer description) 104
6.5 Design 105
6.6 Using the warehouse program 107
6.7 Design review meeting 109
6.8 Coding 110
6.9 Maintenance 111
6.10 Bigger and better 114

Programs Supplied on the Diskette **116**

Index **117**

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

TRADEMARKS OF MATERIAL MENTIONED IN THIS TEXT

1 Introduction

Advanced BASIC Programming is not a contradiction in terms. Traditionally, advanced software has been developed in other languages: FORTRAN, COBOL, and, more recently, PASCAL. BASIC was designed to be an introductory language, not one for high-powered needs. Yet, BASIC can be used for those needs, often as well as or better than the traditional languages.

This text has two interrelated functions: helping you develop advanced software and teaching you concepts of programming that will carry over into professional levels. In general, software quality is independent of the language used, so learning to write good programs will serve you well if you later move into the conventional languages. The great advantage of using BASIC for learning is that the computer you need for it is inexpensive and readily available.

This text is aimed at Commodore BASIC, and most of it can be applied on the VIC 20. This means that you can learn advanced BASIC on a machine selling for less than $200. By comparison, even a terminal to a computer capable of running FORTRAN will cost several times that amount, typically a few thousand dollars instead of a few hundred. Another advantage of basing this text on the Commodore version of BASIC is that it is applicable to all of their machines, from the VIC 20 and 64 through the business machines which, with peripherals, may cost $5–10,000 and offer substantial business capabilities. So, you may learn on an inexpensive starter system, then transfer that knowledge to advanced applications.

You don't have to use a Commodore computer to learn advanced programming in BASIC—in fact, most of the material in this book is relevant to Apple or Tandy machines as well. In addition, the philosophy and principles of programming apply to all interpretive languages. Most of the material on BASIC itself is common to all dialects, even those most distant from Commodore's. But the versions of BASIC from Microsoft (Apple, Tandy, M- and C-BASIC for CP/M) are close enough to Commodore's for most operations not dealing with peripherals to apply directly. Finally, all the ideas and most of the examples should be valuable to you regardless of the hardware you use.

1.1 Prerequisites and purposes

This book is intended to help you learn to write professional software. It is not a BASIC primer, and it assumes that you already know the elementary uses of the language.

The rules for using BASIC are provided with most computers. Other sources of elementary information are books, programs, and courses. For all of those and for other current information, see your dealer—the microcomputer field changes too fast to provide current references in a book!

When you have finished with the elementary materials, you should know enough to write a simple game or to read most BASIC programs. The next step is to read and to write until you understand the mechanics of programming. One way to combine the two tasks is to buy some software on disk or tape and change it. What program you use is almost irrelevant. Any routine can be written in many different ways; its author chose some to write the program, but you can look for others.

Take the program apart from its listing so that you understand how it was built. Think about ways to change small parts of it and try them out. The point is not to get a "better" version (although you may), but to understand how it was written and why. You will quickly discover that it is much more fun to write programs than to run them.

Make sure that you don't start with too ambitious or complicated an example. Tic-tac-toe or a simple slot machine will do as well for this purpose as ADVENTURE, and it will be a lot easier to learn. A good starting point would be an inexpensive game—but make sure you get one that's written entirely in BASIC. The example program in 5.10, DOMINOES, will do on a PET or a 64, although it is more complex than necessary for the job.

After you know how and why simple software is built, you're ready to learn advanced programming. Again, it helps to use an example. Commodore's Word Machine is a good example for most purposes. It displays most of the features of advanced software and is interoperable on all 40- and 80-column Commodore computers. The example program, SUPERLIST, provided with this book may also serve, but it does not use as many of the features as Word Machine.

You need not own a computer to learn advanced software or to use this book. You will want access to one for enough time to convert book knowledge to practice. Each of the sections of the book takes only a few minutes to read, but it should give you enough ideas to spend an hour or more in practice.

While you can write good code for a tape-only system, professional software is normally aimed at disk. First, the disk saves so much time in programming that tape alone is impractical. Even more important is the fact that a serious program will use the disk to extend the computer's built-in memory. A printer is almost a necessity, too; it lets you read listings that would be too complex for a screen and correlate routines from one part of the program with those they call in another. Finally, if you are using a Commodore 64, get a monochrome monitor. You will spend hours reading text on the screen; fighting hash from the modulator, poor resolution from color dots, and blur from a poor television set costs more than the monitor's price of $100 or so.

Even with all due diligence in reading this book and practicing, you won't become an instant programmer. You will learn how good programs are written, how to read and use them, and how to know them when you see them. You will learn what can be taught about writing advanced software; what can't be taught you will have to acquire with experience. Neither this nor any other book can substitute for getting your hands dirty writing programs.

One message is repeated over and over in the pages to follow: there is no "best" way to write a program. The objective of good software is to develop programs that do their jobs well. Programming that is optimal for one application may not even be adequate for another. Users differ in needs, experience, and taste; your programs have to fit all three. Remember your market as you write your program, and don't try to make it be all things to all users. For example, the Word Machine was written to complement conventional word processors in a different market. WordPro, Wordcraft and the rest are superb programs with capabilities the Word Machine does not duplicate. However, using those capabilities requires study, practice, and commitment. They are fine for an office, for an operator who will use them hours per day. But Word Machine was aimed at the home user, someone who will want to use the computer to help write a letter, a book report, or even a book like this one. It was designed for simplicity, especially for the occasional user. It cannot replace WordPro in the office, but WordPro does not fit the needs of the amateur computer user. When you build your software, it should be aimed at *your* user. This book should help you hit that mark.

Professional software is written for both users and customers. The customer is the person who needs the program's products; the user is the computer operator who is responsible for running the program to generate those products. Even when the user and the customer happen to be the same person, it is necessary to keep in mind the difference in needs between the two roles. To facilitate separating those functions, we consistently refer to the customer as female and the user as male in this book. (But no implication is intended that women are less comfortable with computers than men just because customers tend to be less familiar with them than users are.)

1.2 Principles of BASIC

The greatest advantage and the greatest drawback of programming in BASIC is that it is an interpretive language. The computer does not actually execute the instructions you write; it interprets them into primitive instructions it understands, then does what those primitives command. One BASIC instruction may translate into dozens of primitives. Running some of the primitives may take much longer when they come from interpretation than when they are written directly, for reasons we'll cover later. As a result, BASIC runs slower than languages which are compiled, like FORTRAN or COBOL.

The advantage of an interpretive language is that it simplifies program development. When a program is compiled, the interpretation of its high-level instructions is done once, taking much longer than running the program. The resulting code will run very fast, but getting to that point takes compilation time. A program of a few thousand lines may take ten minutes or more to compile, while it would run in only a few seconds. The compiled code might then run in a fraction of a second. So, the trade-off is between speed of getting a program into a position to run (no time for interpretation, minutes for compilation) and speed of running. When you are writing the program, speed of preparation is all-important.

The property of being compiled or interpreted is not inherent in a language. There are interpretive versions of FORTRAN and COBOL, and compilers for BASIC. It is possible to have the best of both worlds: to develop the program interpretively, then to compile the finished product for running speed. Doing both gives any language the advantages of both modes. Interpreters for languages designed for compilation are used in the SuperPet. It is designed to let you develop a program interpretively, then to download it to a larger computer for compilation after you have it working right. Similarly, BASIC compilers such as Petspeed and DTL's give the BASIC programmer the speed of compilation after development. Just to complete the story, some modern languages (FORTH, LISP, PASCAL) use variations on the concepts of compilation and interpretation and offer some of the virtues of each. Since the principal difference between BASIC and the "professional" languages is cancelled by the availability of alternative compilers and interpreters, the operational choice of language must come from other considerations.

Fundamentally, almost any problem can be programmed in any language. However, some problems are better suited to one language than to another. Spoken languages have similar properties; for example, the sound of a poem in French may make its translation into German difficult. Similarly, FORTRAN is designed for mathematical computation; it is less than ideal for creating business reports. COBOL is intended for formatting files and print but is a poor choice for complex analysis. BASIC trades off among the features of specialized languages. It's good for almost all uses, but perfect for none. BASIC is a general-purpose language and can be used easily for almost all needs. It can be used for conventional and fast Fourier transformations which are almost impossible for COBOL, and for banking systems that would tax FORTRAN to or past its limits. The mathematical program might have been better in FORTRAN, the banking system would have been easier in COBOL, but both are possible—and operational—in BASIC.

In practice, the greatest drawback to coding in BASIC is that it has no prestige. After all, it is basic, easy to learn, and at least partially comprehensible to the untrained programmer. A FORTH programmer has credentials established by the simple fact that she can use the language. BASIC provides no such cachet of ability. If you want to write a program, the difference doesn't matter; if you want to be recognized by the drop of a buzzword, you need another language.

1.3 Interoperability

In general, a package of advanced software may be written for a specific target machine or for any computer in a broad class. If you write for one computer model, you may exploit all of its special features and may incorporate machine-language elements freely. There are two drawbacks to that specificity: the code cannot easily be moved to another machine for another customer, and you require a backup machine (to handle hardware failures) with exactly the same properties. On the other hand, if your programs can be used on any machine in a family, you have a broader market for them and easier backup than if you design for only one target. When requirements permit it, interoperability has a high payoff.

There are many dimensions of variation among the Commodore computers: monochrome/color, 22/40/80-column screen, 3/8/16/32/64/96K memory, BASIC 2.0/4.0, etc. Commodore has provided enough commonality in the midst of this diversity to support interoperable advanced software. For most purposes, we may assume a 32K, 40-column, monochrome computer. The program should be able to exploit extra memory if it's available—preferably by dimensioning arrays based on FREe memory. An 80-column screen may be windowed down to 40 most easily by using only its left half. Color can be disdained just by presetting background and character colors. And all of that can be done in such a way that the program can be loaded and run on a 2001-32, 4032, 8032, CBM, 8096, SuperPet, or 64 without the user having to input the computer model.

Unfortunately, Commodore has no cell of memory that can be read to identify the model and its version of BASIC. Fortunately, that isn't necessary when you are programming in BASIC. The old versions of BASIC and the old character generator can be ignored because those machines have been out of production for so long that anyone who still has one should have updated ROM's long ago. In fact, you can count on at least BASIC 2.0 since the earlier versions would not run the disk.

To achieve interoperability for graphics, you use a code which is very simple. On machines with windowing (4032, 8032, etc.) and on the 64, you simply print a chr$(14); on 2001, POKE59468,12. The poke is safe on all machines (except the VIC), and the character does no harm where it is ineffective, so you may do both of them. The 80-column machines need to be windowed down to 40, and any prior window should be cleared first. (A "window" is a restricted area of a screen used for display. By reducing the 80-column screen to a 40-column window, programs may operate on either with a common user interface.) Finally, character color must be set for the 64 (unless you like to read light blue on darker). All of those functions can be accomplished in a single line, using a simple trick of programming.

When you type a cursor-control character inside quotes, you see a character in reverse video. For example, HOME is a lowercase "s". When a Commodore BASIC program puts up a byte of control information in quotes, it uses the equivalent display code in reverse video. As a result, chr$(0) is reverse "@", chr$(1) is "a", etc. That reverse "s" corresponds to chr$(19)—which is the HOME character. You can type the reverse characters directly by entering PRINT, quote, quote, delete, reverse, and the desired equivalent letters. What you have done is to set up the quotes mode, cancel it with the second quote, then delete the quote character. Then the reversed characters are typed in and you don't need to use the CHR$ operation. *Note*: control (27) and return (13) do not work as characters.

For interoperability in graphics mode, we need first reversed "Sss" to clear the screen and any prior windowing. Then "No" to set graphics mode and top left of the window. Now we give 25 "q"s and SPC(39) to get to the bottom right of the window, then "O" to set it, and "e" to print in white on the 64. Finally, POKE59468,12 and the program will run in 40-column graphics on any Commodore host except a VIC.

Getting into literals mode is slightly more complex. The problem is the extra spacing provided between lines on the windowed machines; on a 4032, that hides the top and bottom lines. So, we need the compressed screen that comes from graphics mode, combined with upper and lower case. By using POKE59468,14 (which we need for the 2001 in any case), we accomplish the desired effect on the windowed machines as well. However, we haven't handled the 64. For that, we need to POKE53272,PEEK(53272)OR2. (The nominal value in 53272 is 21 for graphics and 23 for literals, but Commodore doesn't guarantee that those numbers will work on all future hardware. They have assured us that the OR operation will work, so the extra nine bytes are needed.) Lines 9020 and 9025 of SUPERLIST (Section 5.7) provide just that code.

On one expansion system for the VIC, giving 40 columns and extra RAM, the logic works without modification. However, the VIC 20 locks up on POKEs that are safe on other machines, and you will have to play with any adapter you might want to use. The reason for not writing interoperable software for the VIC in the first place is that its limited screen (22 characters) and small memory as supplied are too restrictive for other uses. There are many adapters to get around the problems, but each is likely to need unique code. In practice, cutting an 80-column screen to 40 will win few friends, but there are so many 40's around that writing code for that mode makes sense. Advanced software can and should be written for the VIC 20, and everything except interoperability applies to that machine; in fact, the limited capabilities of the VIC increase the payoff for top-quality programming. But a good program for an off-the-shelf VIC will be poor for an expanded one or for one with 40 monochrome columns; it will be even worse for a 4032 or a CBM. Therefore, regard the VIC market as separate from that for the other machines, and don't strive for interoperability between them.

2 Writing for the User

Advanced software is written for use—and for a user. Usually, your program is intended for a class of user—novice, frequent user, occasional programmer, or serious programmer. A program that gives the first-time computer user simple capability must be self-explanatory, must use simple language, and will trade away sophisticated capabilities for clarity and flashy display. One written for the advanced programmer may do without in-line instructions, may use computer jargon, and may provide extra capabilities that are hard to reach. If you are writing for the data-entry professional, you may use elaborate special functions to provide fancy operations—your user will remember them because they are frequently used.

In writing for someone who will use the program only occasionally, it is better to run a menu than to use special keys for special functions. A menu is simply a screen of options and a request for the user to select among them. One way to select is by a code number; the menu assigns a number to each option, and that's the way the operator reaches the function. That method is easy to code, and it's necessary when you don't know the options until the program is run. For example, if the user will define the categories into which a data item will be placed, selection of categories requires numbering. The category names go into an array, and the index is both the array pointer and the value used in categorizing the information.

Single-character input (see 3.6 Keyboard input) gives you an easy alternative when the programmer defines the options. Use the initials of the options for selection. Pick wording for them that is easy to recognize and to remember. Pick an order for display that is easy for the user to scan. Then put the initials into your working string in an order that's easy for the program. When you're dealing with a nonprofessional user, always adapt the program to your customer. Lay out the screen, take inputs, and provide outputs for easiest use; protect as much as possible against simple errors. For the true novice, don't let any one keystroke cause a catastrophe. There is no better indicator of poor thinking by the programmer than code which bombs when an extra RETURN is pressed.

A general rule for all users is to protect against losing information inadvertently. Before you let the user throw away his carefully written file, verify that that's what he wants to do. That means setting flags when information has been created or changed and when it has been saved. If he asks to input something new when information hasn't been saved, provide a warning and ask for confirmation. Do the same when exiting the program.

It is always worth memory to simplify the user's job. For example, the logical order in which information is entered into a program may not be the best order when the program runs. Even though you have to "waste" variables to take the input out of order, do it. A common example occurs when a program must do a lot of processing and substantial printing. If you take the user's input for both operations at once, when the first begins he can go off to do useful work, leaving the computer to do its part unattended. An occasional check of the screen or the printer should indicate that progress is being made and give an indication of how much longer it will be before he needs to return. If you organize the program for your convenience, the user will have to sit around for minutes or hours, just to hit a key or two every so often. It may save you a few bytes of program, but it won't win you friends—or repeat customers.

It is difficult to remember the user as you program. Take the time to write down what the program will do before you start to code. Offer it to your customer for review, comments, and approval. The most important thing to tell her is what she will see and what she must do to get what results. Talk through the special needs and special features of the problem, and make sure that you know what she needs before you start to code.

Murphy's laws could have been written for computer users. The programmer must expect the unexpected and provide workarounds for everything that can be anticipated. Among the unexpected events that happen with regularity are: hitting the key next to the one wanted, power transient, and disk fault. Requiring confirmation of critical inputs is part of the solution to the keystroke problem. In other cases, you may require confirmation of a set of data before saving, facilitate deletion of individual characters, and provide easy editing of data already filed. You can't protect against wrong-headed users, people who confirm what they mean to deny or those who won't type an "a" to "Add" information, but insist on using a "c" to "Create" it—regardless of the menu. But put in thought and code to keep from penalizing the user for hitting the wrong key. There are programs pretending to teach children that feed back "shame on you" and similar messages if one keystroke is wrong. Remember, if your customer is bothered by your program, she can always get another one.

A few simple rules provide at least a little protection against faults of disks and power lines. Don't leave a disk file open while waiting for input; open it, do what you must, then close it again. The APPEND command is the best way to add to a file that will be needed over a long interval. If your program OPENs once, then uses PRINT# every several minutes for hours, there's a good chance that a day's work will be lost by a simple error. You cannot protect against a faulty disk in all cases, but you can minimize its impact. The simplest solution is to write any critical information to both drives when you can; it then takes correlated errors on both disks to cost you essential data.

A good program is designed for its user, not adapted to him. Your file structures, use of redundancy, and provision for backup should all be aimed at your market from the time you first block out the idea. Remember that it's always easier to rip out features that aren't needed than to add them after the program is built.

2.1 Program planning

This section identifies one method of organizing programs. There are many others, but this approach is efficient and has been the basis for a few dozen applications. So far, it has worked. If you plan to use what you learn on one program to develop another, some system is mandatory. You will quickly discover that the more systematic your programming methods, the easier it is to modify and to maintain your products, and the quicker it is to build on the experience that a successful project can provide. (And don't forget the flops—they may provide more education per hour of coding than your greatest hits!)

The first step in programming is the most important and the most demanding—thinking. The best rule for quality programming is: think first, code later. The necessary minimum set of information to be pinned down before you turn on the computer is:

file structure
major program blocks
utilities required
anticipated problem areas
array structure
global data definitions

A sound file structure is mandatory for microcomputer programming. You will have to trade off speed against storage on the disk and in the computer itself. But even more critical to the long-term success of the program is your anticipation of requirements not well defined when you start. Changes are normal after programming begins, and painfully common even after the system is working. If your file structures won't expand, you won't be able to handle growing requirements. Your product will be limited by your failure to think before you code.

A program consists of collections of lines of code. Good programming practice breaks the total job into major blocks or modules, each of which does just one part of the job. One reason for modularizing is that it simplifies changing the design. Another is that it shortens checkout and debugging. For example, one module may read files, another write them, a third supply all formatted print. Then if the system grows to include a second disk drive or if you change to a letter-quality printer, all corresponding changes are in one place, and they can all be made as a unit.

Frequently, the major functions seen by the user correspond exactly to the software modules. However, there may be logically different operations that are functionally similar and fit within a single module. For example, creating a new record is logically identical with editing an old one—the "old" values are simply nulls. To the user, creation and editing are logically different; to the program, they are the same. The converse can also be true: logically, one might edit a record out of existence; functionally, deletion may be significantly different (e.g., requiring repositioning of pointers, editing of correlative files).

Modularization is the process of blocking out the functional code to correspond to what the program must do. A high-level flowchart is one way to do it, showing the relationship between the user's logical operations and the program's modules. Typically, there will be about six high-level modules in a program, and they can be entered at line numbers which are multiples of 1000. There should be no more than six routines within a module, and each can usually be started at a line numbered as a multiple of 100. Except where speed of execution dictates, all code used by a module should lie within it. It may take a few more bytes of storage to collocate the code, but it is usually worth it in legibility of listings. One exception that can be made safely is exit from a function to wrap up operations or utilities. For example, exiting from a subroutine to the logic for "Hit a key" can be done as a GOTO, rather than a GOSUB followed by RETURN. Similarly, it may be simpler to use a common routine to close files opened for different functions and to return control to the calling function from the close-file logic.

Utilities are packages of code used in many different programs, performing functions common to many parts of each, and with few or no variations among applications. They are standardized subroutines for *your* needs, and give you a library of programming tools almost as convenient as an extension of BASIC. When you organize your program, you identify and collect the utilities that you will need for that job. Utility code tends to fall into two speed categories: very fast and very leisurely. Elements that require user interpretation (e.g., yes or no answer) can be quite slow; they'll still be faster than human reaction times. Code that collects information (even from the keyboard) should be as fast as possible. So utility software tends to be split into two blocks: one at very low line numbers (before the main program entry point), the other at very high ones (immediately before initialization). A utility's requirement for speed comes from the job it does within the application; consequently, a routine that can be run at a high line number in one program will not have to be moved to a low one for others.

When you first think a project through, you should have little difficulty determining where the biggest problems will be found. One area is a system capability you've never used before. Another may be an algorithm you have to invent, particularly one for string manipulation. A third may be trade-off for speed or memory savings. Before you start to code the parts you know how to do, make sure that you have solutions to the tough ones. Develop sample code for a new capability and check it out by itself. Build and test different routines for your new algorithm, so you have several choices depending on what needs emerge as the design develops. Plan alternative designs trading speed and memory, and make sure you understand the amount of effort and the performance payoff for each. It is vital that you keep a record of what you learn as you experiment. Otherwise, when the problem recurs a year or two later you may have to repeat your learning experience. (In general, it isn't enough to copy the solution from one problem over to the next; conditions and requirements change too much to permit reuse without some change.)

In advanced software, arrays form the backbone of the data in the same way that a high-level flowchart forms the backbone of the code. Once you define the data structures, many routines have only one logical design, most specific data elements are obvious, and you can reasonably estimate total memory requirement. If you wish to use a simple routine to input or output a complex file, you may need a multidimensioned array. Then everything you need to know about, say, a telephone book entry goes under one index, with the different fields as a second index.

Preplan your fields carefully: it's tough to split or combine them after you've started coding. Since you probably want to order the information last name first but print it last name last, the name will be at least two fields; if you don't need ZIP code sort, it need not be separated from state. Each field takes significant memory and often substantial time, but the price for too many is usually less than that for too few. The objective of planning is to pay neither price—to get it right the first time.

Global data are variables used to communicate among several parts of the program. They should be named uniquely and never reused in the program. A simple example is the date, a value needed in most advanced software. Once you accept it (with any validation required by the application), assign it to a unique name (e.g., dt$) so that any routine that needs it knows where it is. In planning your program, identify those global variables by description first, then assign names that you can remember easily. Where it makes sense, carry the names over from program to program—if you chose to use dt$ for a banking job, it will work just as well for a word processor. Some variables will become utilities for you:

r$	carriage return (chr$(13))
dt$	date as a string
dt	date as a number (Julian?)

When you standardize information, you can look at an old listing and recognize immediately what you were doing (or trying to do) without having to refer to your documentation. If you remember not to use your own "reserved" words for other purposes in programs that don't need their normal ones, you will find that a year-old program is almost as easy to read as the one you finished yesterday.

2.2 Program organization

Program organization means laying out the modules and their routines for speed and clarity. Almost all executable code is perceived as being better if it runs faster. Obviously, speed is not an advantage for commands that do not execute. In BASIC, those commands are DATA and REM.

The first time a READ is encountered (after CLR, RUN, or RESTORE), the program is searched for the first DATA statement. A pointer is set to that statement, and each subsequent READ advances it from the place to which it last pointed. Given that some DATA statements will occur late in the program, the search will go through all the code once in each program execution. So there's no noticeable advantage in putting a DATA statement early in the listing, and there's a real drawback. When you GOSUB or GOTO a line across the DATA line, the interpreter must jump over it. If that call is executed often, it can waste significant time. So, DATA lines belong after all operational BASIC code. To help in maintenance, it is worth a few bytes of REMarks to explain their functions, but only rarely is clarity worth the time to put DATA statements near their READs.

A REM that ends a line of BASIC wastes some memory, but no execution time. If you're not pressed for space, use REMs freely to explain to yourself and to your user what that line is doing. There are only two cases that justify a line that has only a REM:

entry to a major module
an option that doesn't justify a run-time question

(A "run-time" question is one asked while the program runs. It requires the operator to supply an answer every time the program is used, which is an obvious inconvenience if the answer is always the same. One of your programming decisions is which parameters to build into the code and which to set at run time.)

A line containing only a REM can stand out in a listing. When you're scanning the printout to find a fault, such a line can save you precious seconds—and be worth the microseconds it wastes when the program runs. It may be a matter of taste, but a REMark line for each module is usually well worthwhile.

There are times when you will put a capability into a program to be enabled or not depending on things that change only rarely. For example, most users will have only one type of printer. The two or three terms in your code that depend on which printer is in use can be put into the program and left there, under a REM, to be enabled when the hardware changes. The thirty seconds it takes to "modify" the program for a letter-quality printer can be provided by the user, the dealer, or any competent programmer—if you remember to document what you did.

Initialization is the first job the program does in time—and should be nearly the last thing in its space. It is code that is executed once and includes few addresses. Most of initialization just steps forward line by line; without GOTOs and GOSUBs, its speed will be essentially independent of its location. Therefore, it may be assigned to the 9000's, conveniently low and out of the way. The only code that goes below initialization is error handling that requires user intervention. However big your program is, running all the way through to a disk error trap at 60000 won't take a thousandth of the time the user will spend. He'll be trying to figure out what to do with a "disk error" message; a few milliseconds' delay in getting that message onto the screen will never be noticed.

Low-speed utilities can usually be really slow. They tend to entail human reactions, so computer times are negligible in comparison. They can be tucked out of the way in the 8000's, in whatever order you find convenient. In contrast, high-speed utilities (e.g., GETting characters from disk or keyboard) must be at low line numbers. The only other code that goes into those precious lines is a high-speed processing loop that can't be coded without at least one GOTO into itself. When they can't be avoided, they can be made less painful by putting them at the front of the program.

One function that runs almost as slowly as user interface is the printer. Typically, a dot-matrix printer will put out about one line per second. Program delays from putting print logic in the 7000's will hardly matter on that scale. If you want the code to be interoperable (even with run-time option on which printer is used), printing becomes a common routine which is almost a utility.

Sticking to the idea of starting a module at a line number divisible by 1000, we still have six entries available without crowding. Typically, one will be used for input from the disk, another for output to it. That leaves four to do the program's work. You can start a menu at 1000 or less; putting it at 100 is sometimes awkward, but it has worked well in a lot of applications. It leaves space before 1000 for many quick options, for checking flags (e.g., loss of data), or for updating standard information like Subject or Title. In a word, the space between the menu and the first application module is used for routines common to the start of several user (menu) functions.

Once modules and special routines have been located in the program, design within the module determines both clarity of code and speed of execution. The code is easiest to read if it runs in a straight logical line. Unfortunately, the requirement for speed translates into optimizing the most commonly used path. For example, we might want to check a disk input for end-of-file. The clearest code might be:

```
10 GET#1,C$:IFST=0GOTO30
20 X$=X$+C$:GOTO100:REM END OF FILE
30 X$=X$+C$:GOTO10:REM ATTACH THE CHARACTER
```

That version does the job, but every character except the last takes two GOTOs. If that routine is located at 10, the wasted time may not matter. But if you put it at 6010, it will slow down disk access—even on a floppy. Recode it to speed things up a little with:

```
10 GET#1,C$:IFST=0THENX$=X$+C$:GOTO10
20 X$=X$+C$:GOTO100
```

It's shorter and faster—one GOTO instead of two. If you want still more speed (and are willing to be obscure to get it), try:

```
10 FORI=1TO255:GET#1,C$:IFSTTHENI=255
20 X$=X$+C$:NEXT:GOTO100
```

Now there are no significant GOTO's; the code will run as fast at 6010 as at 10, and a novice programmer will have quite a struggle to figure out why and how the routine works. The "best" organization of this part of the program depends on:

1) How often it must run.
2) How high a line number can be used.
3) How well a programmer will maintain it.
4) How much time you can afford to debug it.

Microcomputers have a fundamentally different kind of timing requirement from mainframes. Even at its cheapest, time on a mainframe is expensive. Time on a micro is (essentially) free. As long as slow operations are done when the computer is otherwise idle and take no operator action, they can run for hours. Put yourself into your customer's head when you organize your program. One way to wrap up a day's work might take an hour, with keyboard activity on ten scattered occasions. Another design might take four hours, but all operator activity takes place in the first five minutes. Count on it, the customer would rather let the computer stay on overnight unattended than spend an extra 55 minutes of operator time. When you organize the program, collect the slow code into one function; the user can go to lunch while they're running. Collect printing when practical; the door to the room with the printer can be closed while it does its thing. And keep a counter on the screen to tell the user that the program is working—and when he has to return.

2.3 Structured programming

Programming has elements of science, technology, and art. By applying jargon, programmers and computer scientists have been able to make folk art look like science in the Nobel-prize class. Professors don't tell you that structured programming has never had a meaningful test. But ''structured programming'' are great buzz words, and can give the uninitiated a warm feeling that someone really knows what's going on.

Structured programming for compiled languages includes a number of features. None of them were new when the term was invented, and some of them are important to the BASIC programmer. For example, a routine coded according to the structuring rules has a single entry and a single exit. Single exit is a good idea when the application permits. If you forgot to close a file, you only have to add CLOSEn in one place; if you want to add a feature, you put it on all exits at once if you do it before a common exit. But sometimes a single exit is very costly. For example, to get out of a nested routine you must set a flag, then test it and use a GOTO around the normal code at each level of nesting on the way out. Within reason, the trouble can be worth the effort.

In contrast, single entry carries a high price and little payoff. A single routine can get any character from the keyboard that fits a string provided by the call, can provide a question mark or not, and can get the answer to a yes/no question—all by entering it at three different points. Structured, the same job would take a collection of GOSUB's or GOTO's. A true believer in structuring would ask for three GOSUB's to answer "y" or "n."

Another structuring rule that doesn't fit an interpretive language is to avoid GOTO's. To do that, we would have to put initialization at the front of the code, and pay a substantial speed penalty for the luxury. Common, high-speed subroutines would have to be put at large line numbers. The slowdown from this rule would make BASIC useless for a lot of otherwise reasonable applications. Interestingly, the kind of "train-of-thought" programming that goes with this rule is the sort that weak BASIC programmers fall into. If you sit down at the computer before you think, your first cut at the program is likely to have no GOTO's at all. Of course, before you get it to crawl (it will probably never run), you're likely to have a lot of them. Typically, a major omission resulting from failure to plan will take at least two GOTO's or their equivalent GOSUB's for its patches.

The remaining fundamental rule of structuring deals with "computed GOTO's." In BASIC, we use ON. . . GOTO instead. The equivalent structuring "rule" would be an explicit test before each ON. . . GOTO (or ON. . . GOSUB) that the variable we branched on was in the legal range. As long as the variable cannot be negative, the BASIC construct doesn't need the test. ON. . . GOTO has a built-in safety feature that we can often afford to use: if the variable is out of range, processing falls through to the next command. So, all you would need to follow the rule is:

```
10 IFC>0THENONCGOTO100,200,300
20 GOTO60000:REM ERROR TRAP
```

Strictly speaking, you should also check that lines 100, 200, and 300 all exist. But you would have done that anyway, wouldn't you?

Structured programming started out as a structured process for developing good programs. The rules were originally illustrative and representative. It has degenerated into a set of absolute rules; the ideas have been lost. In this text, the ideas have been imbedded because they are both logical and constructive to our purpose: writing good code. If you know what to look for, you'll find the philosophy of structure development in Program planning, Program documentation, and half a dozen other sections.

One topic not covered in this book is flowcharts—structured or not. If you find flowcharting convenient, by all means do it. If you think it will help you design, draw the charts before you code. If you think a flowchart helps maintenance, draw them after. But the author's experience is that flowcharts are optional. Data structuring is mandatory. Blocking your code into modules and routines is usually enough to replace flowcharts.

BASIC is so easy to read that diagrams contribute little to understanding the design or its implementation. But writing down exactly what each variable means and what is meant by each byte (or even bit) of a file has a very high payoff. While you write a program, you'll change its flow with every idea. But you can't afford much flexibility in data after you write your first module.

2.4 Program documentation

Just as you must tailor your program to its user, you must aim your documentation at its readers. The problem is that there are three different classes of reader for advanced software: the customer, the user, and the programmer. The customer needs to know what she is buying; the user, how to make it support him; and the programmer, how to adapt it to changing needs. As a result, three different levels of documentation must be generated for each operational program.

Customer documentation is a single page describing what the program does for her. As an overview, it identifies the major functions and attributes of the program. It provides enough information for the customer to verify its satisfaction of her real needs. Customer documentation also constitutes a sales brochure for both the original customer and for others with similar needs. Even if those requirements are substantially different, your previous product is an example of your work and therefore helps in establishing your credentials. Customer documentation is your program's interface with the people who want to buy it.

The first product you generate on any project is the draft customer documentation. As soon as you have blocked out your solution, embody the results in a nontechnical description. One outline for the document is:

Purpose
Concept
Functions provided
Limitations

Make sure that you say enough to demonstrate your grasp of the problem, and enough to let the customer specify changes before you begin development. Final documentation is written when the program is complete; it should be identical with the draft except where comments on the original caused the design to be changed.

The User Manual is the means by which the user accesses all of the capabilities of your product. In microcomputers, it is the requirements specification against which your product is tested and (you hope) accepted. The easy way to generate a User Manual is to write down the hardware requirements, then each step the operator goes through, with each option and its results. For simple products, that kind of documentation may be sufficient. It is always necessary. However, the User Manual written that way serves primarily for training. Its other function, as a reference, requires a different point of view. For training, you tell the user what happens at each step; for reference, you tell him what steps are needed for an objective. The User Manual must provide a map of the functions of the program; when the terrain is complicated enough, recommended routes are needed to get the user from place to place.

Since the User Manual serves as your specification, its acceptance by the customer (with input from the intended operator) constitutes your approval to start coding. It is written to pin down exactly what the customer will get when the job is finished. Since it is comprehensive, it will necessarily include some technical depth. The draft approved at the start of coding evolves in depth and detail as the program is built, but nothing in its initial version should be changed except through coordination with the customer.

Programmer documentation is usually identified as a Maintenance Manual. It contains everything required for a programmer competent at the required level to modify the code to serve new purposes or to correct errors. This document is required for all operational code and for any product sold, even if you intend to do all maintenance yourself. There are two pressing reasons for thorough documentation of your own code:

1) Your customer must be able to maintain the product even if you become unavailable.
2) You cannot expect to remember enough about your code a year or five after writing it to modify it easily.

Maintenance documentation is written after programming is complete, but comes from notes made throughout development. It includes a description of each variable used, detailed file concepts and realization, and nearly line-by-line analysis of each program module. Use of REMarks in the program is not a substitute for the Maintenance Manual; they serve as markers for major blocks, but a substantive, interpreted program cannot afford enough remarks to support maintenance requirements.

The one-page description for the customer should be polished prose that you will be proud to send to potential customers. The User Manual must be legible to the operator of the system and be in presentable form. The Maintenance Manual may be hand scribbled if no one is to read it except you, the author, but that approach carries a lot of risk. If your program is written under contract, the Maintenance Manual really belongs to your customer, whether anyone else will work with it or not. If you sell a copy of what you have developed to someone else, either you must make the Maintenance Manual available (for a fee, of course), or plan to maintain the program yourself throughout its useful life. Otherwise, you will hang your customer out to dry. That will cost you at least one customer, and more if she makes her displeasure known. Don't count on your customer to know enough to ask for the maintenance material, or even to know what to do with it when you turn it over. Tell the customer what it's for, and try to persuade her that keeping it is the only protection against catastrophe.

The example of the Maintenance Manual is a good one to keep in mind when you deal with your customers. In general, they are not computer experts and certainly not as proficient as you are. They should look to you for information on your products, and they will rely on your input—often more than you do yourself. Make sure that each one understands every step. Take time and care to be sure that her expectations and your plans agree. Review the User Manual in draft with your user so that he knows just what you're doing for and to him.

If you don't have a customer, documentation is even harder. You have to play both customer and user yourself. When writing the one-page description, put yourself into the place of the businessperson or homeowner who will buy the product. For the User Manual, try to be the computer operator. In both, you must pretend that you know nothing about programming. Without a real customer and user, you will have to act out both roles if your documentation is to do what it must for your continued success.

Maintenance documentation is the information required to correct errors and to add capability after the program is finished. If maintenance is someone else's responsibility, documentation should be at the level you would want if you had to pick up someone else's program to do the equivalent job. You may want the kind of detailed description provided for SUPERLIST in Chapter 5, flowcharts, and data definitions. If you are to maintain the software yourself, organize and complete your notes and review them to be sure that they are sufficient for you to pick up where you left off—even if it's a year or more later. Adequate documentation is boring, time consuming, and essential for a viable program. (Note that the job isn't finished until the paperwork is done but that the program can be used while you're completing maintenance documentation.) Finally, and at the risk of stating the obvious, keep a copy of everything you deliver. You cannot count on your customer or your user to remember where—or whether—your documentation is stored.

2.5 Testing

There is a truism in professional software that no programmer can test his or her own code. You know what the program is supposed to do, and what the user should do to make it happen. Therefore, you are conditioned to do it right. The program isn't correct until it survives someone doing it wrong. Given all of the above, you will usually find that no one else is willing to take the time or effort to test your product. And, once again, you will have to try to forget most of what you know and play the user.

Testing begins where the user does: with the User Manual. As you read through the draft, make a list of everything that it says the program will do, all of its functional capabilities. (In the aerospace business, that document carries the imaginative name of Functional Capabilities List—FCL for short.) If your draft was truly comprehensive, it would also tell you what the program will do on every type of error. In that case, error handling would show up in the FCL as well. What you need to develop a test plan is a list of all the things the program should do and all the things that the user can do to it that it should be able to survive. The test plan is simply the collection of FCL's and fault-finding into a document, with a schedule for the plan's approval, for generating test cases and procedures, and for running the tests.

A test case is a specific problem run on the program. Each case fully or partially satisfies a collection of capabilities from the FCL. If you draw up a matrix of capabilities and cases, you can show that running all the cases tests all the capabilities. When your program passes all the cases, you're home free.

A test procedure is a specific set of steps which implements a test case. In the ideal world, the user who will execute the procedure can write it; the author of the test cases can verify that the procedures are accurate. In reality, you will probably be writing the test cases and running them yourself; you probably can and will do without the procedures, and certainly won't bother to write them down in their painstaking detail.

If you have a real customer, the best you can expect is to have her verify your test plan against the User Manual. Maybe you can talk her into looking at the test cases and test verification matrix. They will be impressive, whether or not she can read them. Even if she doesn't really review the material, having her look it over will help. At the least, she may find something she thought was in your plans that doesn't show up in the documentation. It's a lot better to find that you have to add something before the program is finished than after.

Which brings up the question of timing. The draft test plan should immediately follow the draft User Manual; since you will probably be writing both of them, you can do them concurrently. Sit down with your customer when they're done—before you start to program—and review them as thoroughly as you can. The temptation to skip writing test cases will be great; demonstrate your dedication and fortitude by resisting. Write each test case down on a separate page. When you run the test, note on the same page everything you observe. Even if you pass the test, you may notice something that strikes you as odd. It may be as simple as a job that runs slower than you thought it would. Every anomaly is a signal that something may be off the track you laid. Check it out before your program is derailed. A problem your customer finds will cost you ten times as much as one you catch yourself.

In addition to your formal tests, play around with the program. Don't follow a set procedure, but hit keys at random to do something unplanned—and to check out how the program handles errors. When an error is found and fixed, take the time to repeat any test which used that code. Leave time at the end of the process for a final step: a complete run-through of every test case on your final product. That's the only way to prove to your customer and to yourself that you're finished.

3 Mechanics of a Program

An advanced program seldom stands alone. It is usually part of a package of software that performs a set of functions. Typically, there will be a program used to set up information for the system, another to perform the regular operations, and at least one more for analysis of errors or anomalies. The first and last categories require more operator skill than that for day-to-day operations; as the programmer, you may be their only user. Those support programs need enough documentation for you or your successor to operate them, but not the full package that goes to your customer. Even after the system is sold off, you need to retain the support software for maintenance and extension.

Divide the system into programs based on the user's needs, not the programmer's. If part of the system requires only reading data and should be accessible to a true novice, put it into a separate program which never writes to a critical file. Don't let the hacker foul up your data base. Daily operations may require a second program, used by a professional who is familiar with, but not expert in, the system's details. Exceptional cases, especially writing key files, should be isolated in another program. Keep dangerous weapons out of inexperienced hands.

A typical application program will take 8–10K of the 30–40K available in the machine. If it's much bigger, execution is likely to be too slow for good results—especially on the 64 with BASIC 2.0 garbage collection. The rest of RAM will fill with data soon enough. Utilities and initialization will take about 3K, so each of the programs in the package will have to fit in less than 8K. Since most of RAM is data, most effort to save memory should be concentrated on the data structures. Saving memory in your code is likely to have less payoff than the same amount of effort optimizing data structures. Modularization will already have eliminated most of the flab in coding.

The principal trade-off of memory and time is usually found in running some code to reduce data storage. That balance is the key to building a good system. You must learn your options and their costs by experience; the following material can only give you some ideas to guide you in teaching yourself.

Using the BASIC instruction set effectively can save substantial execution time and some memory. The most important savings come in efficient use of loops; code that executes repeatedly has more potential for speedup than code that runs only once. After you understand the full effects of ON, GOSUB, GOTO, and FOR. . .NEXT, examine the way you use them. Then move modules around, assign routines to small line numbers, and switch among the instructions for best results. Where the code runs frequently or requires operator attention, spend extra effort on saving time. Where it runs seldom, concentrate on saving space. And remember to write down your experience as you acquire it. You'll need it on your next project, and you'll find it faster to use what you learned than to learn it over again.

3.1 Line numbers

A BASIC program is a sequence of lines of code, each distinguished by its line number. When you type in a line, BASIC recognizes it as part of the program because it begins with a number. A command begins with a letter. In fact, that is why a variable must begin with a letter; if you had a variable "2A," the computer would interpret the command "2A = B" as a line "2 A = B." And that is *not* what you meant at all.

BASIC knows where its program begins and has a marker for where it ends. In the early generations of Commodore machines (PET and CBM), BASIC always begins at decimal 1025 ($0401, using the conventional "$" to indicate hexadecimal). Later machines have a pointer to a variable start of BASIC; the interpreter reads it to find out where to begin. The first two bytes of a line of BASIC are the address of the next line. The next two are the current line number. To get the line number from the stored value, first convert the hex to decimal (if necessary), then multiply the second by 256 and add it to the first. For example, if your first line number were 300, peeking at 1027 would show 44 ($2C) and 1028 would show 1 ($01). When you type in a new line nnn, the interpreter scans all line numbers to find the last one less than nnn and the first one greater than nnn. If nnn is already there, it is replaced by the new one. If there is no command on the line you enter (numbers only), then the line is null and it is deleted from the program. If BASIC cannot interpret the line number (<0 or >63999), a "syntax error" is reported. When a line is inserted, changed, or deleted, all lines with higher numbers are moved to make room without leaving empty space.

The first pair of bytes in a BASIC instruction is the absolute address in memory of the next larger line number of the program. When the program looks for a line number, it jumps from one line to the next by using that pointer. The computer knows three important program addresses: start of BASIC, start of the next line, and where it is currently executing (within the current line). When it sees a GOTO command, the interpreter checks whether the high-order byte of the line number is greater than that of the current line. If it's greater, then the interpreter starts looking for the destination at the next instruction. Otherwise, it begins at the start of BASIC. When it finds the line number you commanded, it makes that the current line and continues execution at its first command. (If it can't find the number, you get the error message.)

The GOSUB command is similar to GOTO in its operation, except that the computer remembers where it came from: the address of the next command after the GOSUB is saved in a reserved area of memory called the "stack." Since the stack is limited in size, it can have more information supplied than it can hold. When that happens, you receive an "out of memory error." In this case, that message doesn't mean that you have used up all of the memory, only that you have exceeded the space allotted to the stack. The limit on how deeply subroutines may "nest" (call other subroutines) is established by the size of the stack. When a subroutine RETURNs, the address to which it goes is the one at the top of the stack. In that process the return address is "popped" (removed) from the stack. The space is then available for another subroutine call. If you don't RETURN from the call (for example, if you GOTO an error routine or directly to a menu), the addresses of the current call and all its unpopped predecessors are left behind, wasting space. Do this a few times, and the stack fills up—for an "out of memory error."

It turns out, then, that we know a lot about good programming in BASIC just from looking carefully at line numbers and what they mean. First, we know that we need to RETURN from each GOSUB and why we may get "out of memory error" when there's still a lot of memory free. Second, we have insight into making programs run faster. A substantial program spends a lot of its time going to other locations. If we have it go either to very early lines or to lines with larger high-order bytes, it will get there more quickly. In a large program, from line 2540 (high-order byte $09) you may GOTO 20 (high-order $00) or GOTO 2600 (high-order $0a) very much faster than you can GOTO 2550 or GOTO 2530 (both with high-order address byte $09). As a side note, you can also see why you may not want to renumber a finished program to save space: it could slow things down!

3.2 Commands

A BASIC command is a reserved word which is interpreted by the computer to cause specific actions. Like a line number, a command is not stored the way you type it in (or the way it is displayed when you LIST the program). When you hit RETURN after typing in the line, the computer scans the line for recognizable commands. Each is converted to a "token" (a number between 128 and 255) for storage. Everything else is tucked away in the form the interpreter finds: as unreversed characters (ASC < 128). Since commands cannot be in quotes, the meaning of a token in quotes is different, and reserved command words in quotes are not tokenized.

The exact word transformed into a token is important and is often a source of confusion. Logic was not the controlling factor in defining keywords (or BASIC commands). So, the command to provide a number of spaces includes its left parenthesis ("SPC("), while most others that need the parentheses (e.g., STR$, SIN) do not include them. In some cases (e.g., FRE), reasons for seeming illogic may show up with time.

One anomaly in Commodore BASIC gives insight into the tokenizing process. Within a REMark, tokens are not created. But the stored information is displayed as though it had been tokenized. A shifted letter (a capital in literals mode) is stored as a character whose value happens to lie in the range of command tokens. For example, "A" is character 65 + 128 = 193. If you put that letter into a REMark, the 193 is stored (one byte). When you LIST the line, BASIC translates character 193 into its command, ATN. Now, if you put the cursor on that line and key RETURN, the line will be reinterpreted. The three characters that form "ATN" will be stored separately as three bytes. There are few applications where this process is useful, but it is worth exploring just to be sure that you understand the token concept.

Another form of shorthand commands is useful in programming. One case is well documented, the use of "?" for PRINT. The general case stems from the fact that the BASIC interpreter recognizes a shifted character as ending a command. Therefore, a shorthand form of most commands can be entered by typing the first couple of letters with the last capitalized. For example, the PRINT# command may be spelled out, but its shorthand form is short and easy: pR. To get a disk directory with two characters, use CATALOG in its shorthand form: cA. (DIRECTORY takes three: diR. Shorthand for DIM is dI.)

Note that a line number used in a GOTO or GOSUB is not a command. It is stored as a sequence of ASCII characters, one for each digit. If you're trying to squeeze out the last few bytes of storage, remember that RETURN takes only one byte, where GOTO1000 takes five (six if you put a space before the number). Colons and spaces used to increase readability cost memory; so do REMarks. They may be so important to your purposes that you use them extensively, or you may minimize them if you're running out of space. Of course, a compiled program removes all nonexecutable material, including spaces and REMarks; if you plan to compile, use those features freely to simplify maintenance.

To summarize, a line of BASIC begins with two bytes of the line number followed by the two-byte address of the next instruction. Then comes a sequence of commands (values > 127) and characters. The line ends with a byte set to zero. The overhead to put a line into a program is five bytes; the overhead to add the command on the same line (with a colon) is just one byte. Figure 1 illustrates the storage of a line of SUPERLIST. At the top is BASIC line 1060 as LISTed. By entering the monitor, we find that it stored at \$0739. The first two bytes (\$52 \$07) point to the start of the next line at \$0752. The next pair (\$24 \$04) are the line number, 1060 = 4*256 + 36. The next byte (\$8b) is the IF command. \$51 corresponds to "q," \$46 to "f," the name of the variable "qf." \$89 is GOTO and the next four bytes are the target line number (1400) in ASCII. \$3a is the colon and \$8f the REM command. The remaining bytes are the REMark itself, including \$20's for the spaces. The last character (at \$0751) is the \$00 that marks the end of the line.

```
1060 ifqfgoto1400:rem in quotes
.m 0739 0751
.: 0739 52 07 24 04 8b 51 46 89
.: 0741 31 34 30 30 3a 8f 20 49
.: 0749 4e 20 51 55 4f 54 45 53
.: 0751 00 6c 07 38 04 8b 43 b1
```

Figure 1: BASIC as it is stored.

To save storage and time, put as much as possible on a single line. To increase readability, put separate commands on separate lines. Trade-offs among speed, storage, and legibility are normal in programming, and you should make the choice based on your application rather than on a rigid set of rules. Since most of the things you want to program can be coded in many ways, there may be many "right" answers to any problem. As long as your choice is one of them, don't worry about whether it is the "best" by anyone else's standards.

3.3 Variables

A variable is a thing you want the computer to remember. It has three key elements: its name, its address (location in memory), and its value. Your program controls names and values; BASIC assigns locations automatically. When the running program finds a variable name, it looks for it in the list of those it encountered previously. If it doesn't find the name, it adds it to the list with a default value. From the first time that the name is used, it always has a location and value associated with it; you can see the value (for example, PRINT it), but its location is invisible.

There are three types of variables: numeric, integer, and string. The range of numeric variables is immense: about 35 decimal digits plus sign. Precision is limited to eight or nine digits. A numeric variable is stored as seven bytes of binary information. It is used in binary form for arithmetic, translated into decimal for output. The name of a numeric variable is an alphameric string beginning with a letter and including no reserved word (BASIC command or predefined variable).

Two numeric variables are treated by Commodore BASIC as different if they differ in either of the first two characters. The predefined variable TI can be printed by the command PRINT TIME, since BASIC regards TI and TIME as identical. On the other hand, X, X1, and XI are different variables. Since the size of a numeric variable is fixed, it can be stored in a fixed location with its name; when a different value is given to the variable, the corresponding bytes just replace the old ones in the same place in memory.

An integer variable is logically different from a numeric one but is handled in essentially the same way. (All arithmetic is done in floating-point, so integer variables slow things down a bit.) An integer is limited to values that could be represented in two bytes; its range is from about – 32K to + 32K. Integer variables are named just as numeric ones are, except that they end with a %. Note that X% is not the same as X, and both names may be used in the program without confusing the computer. (However, they may confuse the programmer if they are truly independent variables. You may want to use a convention that X% is used only for the integer part of X; then confusion is unlikely.) An integer variable is stored as seven bytes (but see 3.4 Arrays) and associated with its name. The easy way to think about simple integer variables is that they are another way to write the INT function: X% = Y means exactly the same thing as X% = INT(Y). Like a numeric variable, an integer is set to zero when it is first encountered.

A string variable is substantially different from a numeric or integer variable. The most important difference is that its size is determined by execution of the program, not preset. Consequently, BASIC stores the address of its value with the string's name, not the value itself. (Any address takes two bytes, so needs a predictable amount of storage.) If the value of the string is in the program itself (e.g., X$ = "ABC"), then the address is the location of that statement in the program. When practical, the value is not stored twice. The same thing applies if you READ a value from a DATA statement. Note that a string variable has the same naming rules as the other types (ending with a "$") and that it is initialized as a null string (zero length) when first encountered.

When the value of a string variable is assigned through an operator (e.g., X$ = X$ + "A"), a place must be found to put it. Within the memory of the computer, space is taken first for BASIC's work space, then for the program itself. In some machines, screen memory comes into the picture as well; in the older models, it is up above the rest of RAM. (The screen never enters BASIC's "map" of memory significantly.) At the end of the BASIC program, variable names and values are allotted (or names and string addresses). When you modify a line of BASIC, you may move the place where the variables start; that's why all variable definitions are cancelled when you change the program. When a string is assigned through an operation, its value is stored away in the space between the top of variable definitions and the top of available RAM. In fact, the strings are put in from the top down. When the next assignment would overlap variable definitions, it is necessary to free up more space if possible. Since each assignment takes a fresh chunk of memory, most of the top space is filled with outdated information. Those old values are "garbage" to the program; the process of reassigning string memory to discard them is called "garbage collection."

Under BASIC 2.0 (the version in VIC and the 64), that process takes a noticeable time; BASIC 4.0 uses more memory to save that time. In extreme cases (which you are unlikely to find in practice), 2.0 can take an hour and a half to do a job handled in a fraction of a second by 4.0. How frequently garbage collection occurs depends on how much memory is used by the program and variable definitions and on how frequently strings are assigned; how long it takes depends on how many strings have to be collected. If garbage collection is a problem in your code, one method for speeding it up may be to fool the computer. For example, first read in all fixed data from a disk file, then POKE the value BASIC uses for top of memory down under the space they use. When garbage is collected, the space used by the external inputs will not be included, and substantial time may be saved. Many other methods can be used (depending on the computer you have), but try first to live with the delays.

3.4 Arrays

An array is a collection of variables of any type which are distinguished by number (ordinal) instead of by name. The principal reason for using an array is to operate on its elements by calling them with a number, usually in a FOR. . .NEXT loop. The rules for storing information in arrays parallel those for simple variables except that values in an integer array are stored as two bytes each instead of seven. That memory saving is the principal reason for having integer variables and the other reason for using arrays at all.

The first time that an array name (a variable name followed by a left parenthesis) is encountered, a dimension is assigned to it and memory is allocated. If the first use is in a DIM statement, the dimensionality is given by that statement; otherwise, it is given a default assignment of eleven entries (0 through 10). Since memory is assigned to the array when it is first encountered, redimensioning during execution would require moving all subsequent assignments. Such an operation would be slow and would take substantial code, so it is not permitted (redimensioned array error). Normal procedure is to use DIM statements in the early part of initialization. Arrays are stored right after the program and before simple variables.

If you define an array after defining most of your variables, all of the variables have to be moved and time is wasted. That's usually not much of a problem and neither is the ordering of the variables themselves. When BASIC runs, each reference to a variable causes the list to be scanned for that name. If the most-used variables were named early in execution, that scan is faster than if they were named late. However, programs rarely use enough different variables for the delay to be significant in practice; assigning variables early seldom saves enough time to be worth the effort.

Arrays may have up to four dimensions. BASIC stores values and addresses in an order computed from the values of those dimensions. It accesses them by finding the start of the array, then computing the address of the entry you want from the values in your reference. The amount of storage required for an array is three bytes for its name, then two bytes per entry for string or integer, and seven bytes for numeric. Remember that there is always a 0 element to the array, so DIM X(5) means 6 elements of 7 bytes each; X%(4,2) means $5 \times 3 = 15$ elements of 2 bytes each. Two-dimensional arrays are common in advanced programs, while three and four dimensions are needed rarely. For example, a mailing-list program might have one dimension for the fields within a record (last name, first, street address, etc.), and the other for the individuals whose information is on file. Some mathematical problems do require four or more dimensions; if you have one that exceeds what BASIC will handle, it will probably be too big to store in a microcomputer anyway.

One special case that gives a bonus "dimension" occurs often and can save memory. Suppose the information you need to access is always representable as a positive integer less than 256. It is then also representable as a single, legal character in BASIC, and you can map between the number and the character by using the STR$ and ASC functions. Now you can make one "dimension" of a complex array be position in a string (maximum value, 255), and can put that string into an array which may have up to four dimensions. Access the appropriate string in the usual way, then access the extra dimension with the MID$ function (remember that ASC returns the ASCII of the first character but that it bombs on a null string). You may want to use the string even when you have enough dimensions available since it takes only one byte per entry (instead of two or seven). But it is not recommended if you need to do much arithmetic on the elements or if garbage collection delay is a problem. This "trick" is of real but limited use. It has a side benefit since it is not predimensioned. The size of the dimension is simply the current length of that string, and may vary as the program runs. In some cases, that memory saving alone may be worth the trouble of using string length as a dummy dimension.

3.5 Bits, bytes, characters, and numbers

To understand programming, it is necessary to understand what the computer knows as well as how it works. A program puts the contents of memory into a context—establishes their meaning by their use. Therefore, we look into what is stored in the computer and what it can mean to learn how to use and modify those meanings.

The fundamental piece of information is the bit. The least possible thing we can know about anything is its existence. We can call it true or false by establishing a context (if x exists means that x is true). Similarly, we can give it symbols to mean existence or nonexistence, being true or being false. If we want to operate mathematically, we can assign the symbol "1" to existence, "0" to nonexistence. In computer jargon, we call a thing with values indicating truth or existence "flags." Each flag corresponds to one "bit" of information.

When we do arithmetic, we can use true/false (binary) logic. Since that logic has only two values available in a position, it is called "base 2." We assign one bit to each position of a power of two in the number. The first position is 2↑0 = 1. If a symbol "1" is in that position, it means that the integer is odd—has a remainder when divided by the next higher power (2↑1 = 2) even after any lower terms have been subtracted. It is inconvenient to talk about bits one at a time, so we conventionally collect them into "bytes" of eight ordered bits. With eight bits, each byte can count from 0 through 255 (2↑8 – 1). But that byte is not that number; it is simply a pattern of bits which can be interpreted as that number if we wish.

The decimal equivalent of any byte stored in memory can be read by PEEKing at it. The byte may be used as that number, or as a character, or as a collection of flags. It depends on the context established for it by the program. In fact, a byte may assume many such meanings in different parts of the program, even if it always contains the same pattern of bits.

Suppose that a chr$(193) is stored somewhere in memory, and we wonder what it "means." If it is part of a BASIC program, it means the ATN function. If it's in a quoted string, it means the character "A." Or you could have stored it in a string to mean its ASCII value—193. If you use that location as a hex number, it would be written "$C1," which is the meaning that a monitor would show you. As a part of machine code, it could mean an indirect compare against the X register. In binary, 193 means 11000001; interpreted as eight flags, it means on, on, off, off, off, off, off, on. Obviously, there is no inherent or "right" interpretation of that 193. It really means what your program makes it mean.

Two of the meanings of a byte are translatable by BASIC commands. Suppose that the byte is a character, c$, in the context in which it was stored. Then it can be converted to a number by the ASC command. A number in the range 0 through 255 can be converted to a character by the CHR$ command. Every value in the range corresponds to a character (although not all can be printed); every character has a numeric equivalent. What the byte "is" depends on how we use it; its only independent existence is as a pattern.

Changing contexts for information is a common function of advanced software. One purpose is to reduce memory demands. Suppose an item can be put into any combination of seven categories. You may save its categorization as seven separate floating-point ones or zeroes. Since storing a floating-point number takes seven bytes, you would need 49 bytes to hold the information that way. Another way to hold the information is by forming a single number out of the ordered set of seven. Number the categories from zero through six. Multiply the one or zero in the corresponding position by two to that position's number. Then add those products together. The result is a unique number between 0 and 127 (inclusive). So we can store those seven flags as a number, using only 7 bytes. But that number also corresponds to a character, which can be stored in one byte; we can get the character by using the ASC function. (Some of the characters less than 128 cannot be INPUT by BASIC. Therefore, it is often desirable not to use the eighth bit and to store only seven flags per character. The easy way to do that is to preset the highest-order bit to 1 by ORing the number with 128.)

Notice that many operations are required to save 48 bytes of storage, including exponentiations, multiplications, and additions. If we "pack" the data (compress them), we save memory and spend time; if we want to save time, we must use memory. That is a fundamental trade-off in programming. The "right" answer depends on the program's particular requirements. In many cases, the difference between a good program and an unacceptable one is as simple as packing and unpacking in the right places.

The context of information goes beyond the byte level. A floating-point number can also be converted to a string with the STR$ function. The STR$ creates a string whose first position is a blank (chr$(32)) for a positive number, a dash (chr$(45)) for a negative one. The remaining characters in the string are those corresponding to the value in the decimal representation of the number. For example, the number 17 converts to " ", "1", "7." The floating-point number 17 takes seven bytes; its string form stores in three.

Operations on strings and numbers are both logically and functionally different. Adding two numbers does what you expect; "adding" two strings attaches the second to the end of the first ("concatenates" them). In many applications, it is better to manipulate strings than their numeric equivalents. Using the VAL function (the inverse of STR$), you can multiply an integer by ten with:

```
y=val(str$(x)+"0")
```

Sometimes you may work with numbers as strings to save time; more often, you will do it to handle integers of more than nine digits. Another reason for converting from numbers to strings is to facilitate displaying information. We want to put commas and dollar signs into displays for easy readability; that requires treating the numbers as strings.

3.6 Keyboard input

One way to feed characters from the user to the computer is with the INPUT command. For some purposes, it is good enough, and it is always convenient. But advanced software usually needs to use the GET command for one or more of the following reasons:

- INPUT can drop you out of the program.
- INPUT of a string does not check values.
- INPUT of a numeric variable checks too much.
- Miskeying can give strange results, such as including the prompt in the INPUT variable.
- The longest INPUT variable is shorter (never more than 80 characters, less than 39 on a 64 with a prompt) than can be accepted with GET.

Since GET will be used frequently in advanced software, it makes sense to define utilities for its applications. The part that runs at high speed accepts a single character. One version takes three lines:

```
20 GETC$:IFC$GOTO20
25 GETC$:IFC$=""THENPRINT"<underscore><backspace>";
   :GOTO25
27 PRINT"<backspace>";:C=ASC(C$):RETURN
```

Line 20 quickly and efficiently empties the input buffer of any characters left by previous keystrokes. Seeing how this line is written indicates the skill of the programmer. If the test is:

```
20 GETC$:IFC$<>""GOTO20
```

then the programmer hasn't thought about the code but has just put it down. No string can be less than null, so that part of the test is useless. If it's written:

```
20 GETC$:IFC$>""GOTO20
```

then the programmer has not really learned what the test for string equality means. If you GET a character and none is there, then the string is functionally undefined—in a sense, it doesn't exist so the IF test is sufficient with no comparison. Note that using a counter to empty the buffer (it's only ten characters long) is practical but wastes some time and memory:

```
20 FORI=1TO10:GETC$:NEXT
```

Again, it shows a programmer who hasn't thought the job through.

The next job is to wait for a character to arrive. In the process, a dummy cursor is useful, and the underscore is a convenient one since it need not be flashed. If you want a different cursor, just define your own and flash it if you wish. Using the underscore in line 25 allows us to backspace over it to keep position on the display. When a character does arrive, we must blank out the cursor before returning. Since many elements of the code to follow will want the ASCII value of the character, it is convenient to provide it within the mandatory third line.

Users of the routine may or may not want to empty the keyboard buffer. For example, if you are getting a long string, you may want to let the buffer help keep the program's speed up while you're off processing inputs already received. For that purpose, you may enter the routine at 25 instead of 20. But don't forget to empty the buffer whenever you want a new input; don't penalize the user for resting a hand on the keyboard.

The simplest use of the routine to get a character is to get any character as a signal to proceed with processing. This routine need not be fast, so it is dumped down in the 8000's with the rest of the low-priority utilities.

```
8010 PRINT"Hit a Key to continue";:GOTO20
```

is one easy solution. (Use reverse video, color, cursor-down, or other display features for emphasis if you wish.) Note that you don't need to GOSUB20:RETURN. The return at 27 will get you back to the call if you just GOTO20, saving a couple of bytes of program.

Another utility routine gets a single character from the keyboard that suits the program's needs. Define x$ as the string of acceptable characters. The following code will do the job.

```
8100 GOSUB20:FORI=1TOLEN(X$)
8110 IFASC(MID$(X$,I))=CTHENJ=I:I=256
8120 NEXT:IFI<257GOTO8100
8130 RETURN
```

There are many alternatives, but it's worth seeing how this one works. Lines 8100 and 8110 work together and can be combined if you wish. First a character is accepted from the keyboard through the routine we already know. Then the character is compared with the string of acceptable ones by using the ASC function; an alternative is:

```
8110 IFMID$(X$,I,1)=C$THENJ=I:I=256
```

If the input (c$) is accepted, the loop index (i) is set beyond the upper bound of the loop (no string can be longer than 255 characters), and a variable (j) indicates which character was matched. The test in 8120 tells whether a match was found; if not, the loop is reentered to wait until an acceptable character is supplied. If the input matched, the utility returns to its call with the character (c$), its ASCII (c), and the pointer (j) to which value was received. The last is very useful for programs which follow with branching (onjgoto) or which will use the input as the index to an array (x(j) =). It makes it easy to avoid the slow, wasteful code that comes with a collection of lines that read like: ifc$ = "x"then . . . .

There are many bells and whistles to be added to this utility package. The simplest is to print an acceptable character:

```
8130 PRINTC$:RETURN
```

—it takes just four extra bytes. Several extra lines are needed if you want to confirm the input with a RETURN or to cancel it with DEL; whether you want to let single-character input be accepted without a RETURN is your design choice, but you should make it once for a package of software and stick to it throughout. Whatever you teach your user (or yourself) to do in one case, you should do in all. Otherwise, you have to remember which set of rules applies to which program, and that hassle has no payoff.

There are a couple of conveniences that simplify using even this utility.

You will usually be asking a question when you are using the GET package. So, just add

```
8090 PRINT"?";
```

and enter at 8090 to get the question mark and space. The most common answers you will want are "y" or "n"; it's probably worth a line:

```
8050 X$="yn
```

Then GOSUB8050 without having to define x$ for every call.

Another utility program of value is one which GETs a string of maximum length m; that takes about four lines using both 20-27 and 8100-8130. However, if you need to restrict characters in the string (for example, eliminate delimiters such as comma and colon), you probably can't afford the time to search x$ for each character. One solution is to use an array of acceptable characters and a replacement for 20-27 that only returns when the character fits. Note that the whole process is a variation on the theme of using a string as an extra array dimension (see 3.4 Arrays).

Finally, it's worth noting here the very special properties of the shifted space CHR$(160). It always prints as a space, but is not eliminated when leading blanks are dropped (e.g., on INPUT and INPUT#). Therefore, it can be the only character in a diskfile record, where you can still INPUT# and yet have "nothing" in the field when printed. However, a shifted space *must not* be used in a diskfile name; it does nasty things to the directory and may hide the file out of reach. Whenever you specify acceptable characters, give an extra thought to 160.

3.7 FOR. . .NEXT loops

The FOR. . .NEXT loop is one of the most powerful constructs in BASIC. Its functions can be performed with conditionals and GOTO's, but at significant penalty in speed and memory. The index of the loop must be a simple floating-point variable. The size of increment (STEP) defaults to 1. And the user needs to apply care with loops as with subroutines to avoid "out of memory" and "next without for" errors. Those restrictions aren't limitations in practice, and skillful handling of loops will speed up your code.

Let's look at two equivalent pieces of program.

```
10 FORI=0TO9:X(I)=I:NEXT
```

uses a very simple FOR. . .NEXT loop to initialize an array.

```
10 I=0
20 X(I)=I:I=I+1:IFI<=9GOTO20
```

uses the conditional IF and GOTO in 20. Of course, the second example takes more code and an extra line. It also needs a GOTO on each iteration. Late in a large program, that will run very slowly. The FOR. . . NEXT loop saves the location of the command after FOR (and STEP if used) on the same stack that remembers where subroutine calls come from. So, when the NEXT test passes, the loop jumps directly to the appropriate command—no searching for line numbers, no wasted time. When the loop completes (i > 9), the calling information is popped from the stack and execution proceeds.

Any of three mechanisms will keep the stack clean:

complete the loop
reuse the index as an index
RETURN from a subroutine that included the loop

When BASIC executes NEXT, it increments the index by the STEP. If your code assigns a value to the index during loop operation, the increment is applied to the value you assign. Therefore, instead of letting the loop complete naturally you may force completion by assigning a number at or beyond the upper bound within the loop. To find a "7" in an array of n elements, you might use:

```
10 FORI=0TON:IFX(I)=7THENJ=I:I=N
20 NEXT
```

and get the array pointer in j. You should preset j so you can recognize if the test was unsuccessful; as an alternative, you may use i = n + 1 in 10, then test whether i > n + 1 (array element found).

The example problem may also be handled by:

```
10 FORI=0TON:IFX(I)<>7THENNEXT
```

and a test on whether i > n (no element = 7). That code leaves the FOR . . .NEXT information on the stack, where it threatens to foul things up. However, if the next loop following a hit uses i as its index, the old one is popped by BASIC. Even more useful is the fact that a RETURN from a subroutine that leaves an open loop will pop the stack all the way down to the call—cleaning up the debris of the open loop in the process.

As a matter of coding convention, some simple rules are in order. A FOR. . .NEXT loop never needs the name of the variable in the NEXT statement; if you put it there, it is for readability only. The only loop the program can increment at the NEXT is the top one on the stack. That loop has an index, and that index is the one that will be STEPped. If you specify NEXTj and the index is actually i, you get "next without for" error. If you don't specify, i will be stepped—whether that's what you wanted or not. On a single-line FOR. . .NEXT loop, naming the index in the NEXT really doesn't clarify things very much and does take significant time—BASIC has to look up the index to verify that you named the right one instead of just incrementing the one that's there. When the loop takes several lines, the extra legibility provided by the name of the index is usually worth the time. In addition, having the computer check that you are incrementing the right index is worthwhile during checkout of the program—it can help you avoid very weird problems that you might be unable to recognize otherwise. If you are using nested loops, remember that NEXTj, i is convenient, takes one less byte than NEXTj:NEXTi, and does exactly the same thing.

3.8 Machine-language interface

There are occasions when a few lines of machine code will substantially aid operations in your BASIC program. Other times you will want to know why or how someone else's program is doing all those strange things. So this section and 5.8 SYS CHECKSUM are aimed at embedding machine code in BASIC programs for useful purposes.

One major aspect of interfacing machine language with a BASIC program is where to put it. From the start of BASIC to the top of RAM, everything is (or can be) used by your program and its strings. On dual-tape machines the 160-character buffer for the second datasette is a favorite spot for machine code. That space isn't available on all machines, and where it does exist it is used for disk commands as well as second cassette—it's no longer as attractive as it was in the old days. The (first) cassette buffer is available and is a good location for machine code if you aren't going to use tape at all. For larger chunks of machine language, your program may lock the top of memory away from BASIC by appropriate POKEs, telling BASIC that you have a 30K computer instead of one with 32K. That code is highly dependent on which machine you're running, so check a memory map to find out what locations define top of RAM. Note that you must lock that space out before BASIC stores any variables into it; the best bet is to do that before any other part of initialization.

There's another useful spot in which to put a short machine-language routine: at the beginning of your program. There are only two places where BASIC normally starts in Commodore machines: 1025 ($0401) or 4097 ($1001). It is fairly easy to check which one is in use. One way is to print a couple of characters to a clear screen and to PEEK for them where the screen belongs. If your host is a 64, PEEK for start of BASIC; otherwise, it will be at 1025. The first few bytes after that location are address, line number, and your GOTO9000. After that, you may insert a REM and a batch of idle characters. From 11 bytes after start-of-BASIC you are free to have a REM of, say, 60 bytes. Replace them with your machine code by POKEs at any point in the program. That's ample room to do a lot of good machine-language work—exclusive OR, case conversion to true ASCII, or other functions that are faster enough than BASIC to be worth the trouble. For an example of embedded machine language in practice, see Section 5.8 SYS CHECKSUM.

Two commands interface BASIC to the machine-language routine: SYS and USR. SYS is, effectively, a subroutine call to its argument. Any variables to be input to that routine must have been POKEd into appropriate (and safe) locations before the SYS; those returned must be PEEKed from safe spots. Normally, you would use SYS when there are no arguments required. For example, a SYS is used to extend BASIC with machine language in programs like WEDGE and BASIC AID.

USR is a subroutine jump to a reserved area of RAM ($00 on PET and CBM), where you have three bytes to jump to your routine. Its argument is a value sent to and returned from the floating-point accumulator, not the address. USR may be convenient when you want to call a single routine frequently with a variable argument, but its use of the floating-point accumulator makes the machine code awkward for simple applications.

The lack of a true RESET on Commodore machines makes machine language more of a risk than on other computers. If you POKE around in memory, you may lock up the whole system to the point where you must shut off power and start over. (On some machines, you can do physical damage as well; video circuitry may not survive the speedup that can be POKEd into the logic.) Within a BASIC program, a small piece of machine code can save time and confound meddlers.

3.9 Odds and endings

WAIT and RND are useful commands with some odd properties. NEW, STOP, and END all terminate a program, but with interestingly different features. Those five commands really have little in common—except that they are all covered in this section.

WAIT commands the system to suspend processing until the XOR of the address in the first argument and the number in the third, when ANDed with the second, is not zero. It is only useful when the address is associated with an input; since those addresses vary among the types of computer, any program using WAIT is almost certain to run on only one machine. A good application of WAIT and a short way to get a character from the keyboard is to POKE the buffer counter to 0, to WAIT for it to be non-null, then to GET the character(s). For safety, it is best to use:

```
WAIT addr,255,255
```

although it is unlikely that keys could be pressed fast enough to be a problem if you just wrote

```
WAIT addr,1,1
```

instead. A fatal error will result if you WAIT for a cell of ordinary memory; if it didn't pass the test the first time, it never will.

RND is used to generate a sequence of pseudo-random numbers. When the program runs for your customer, randomization is used to get an unpredictable sequence. Unfortunately, that unpredictability makes checkout difficult. When the argument of RND is zero, most Commodore computers will give a truly random number (derived from the clock). For checkout, where you need a repeatable sequence, give your first use of RND a fixed, negative argument. Every other use should be a positive number. When the program has been checked out, change the negative number to zero or to −1/TI to get a random sequence. (The argument can be −TI, but inversion gives better statistics.)

STOP interrupts execution of the program and reports the line number at which the break occurred. END does exactly the same thing, but prints no message on the screen (except "ready"). You can resume operation in either case with CONT. For debugging, STOP is the command of choice. When the break occurs, you know what point you reached and can print out the information you need. Take care not to destroy the information you need to CONT successfully; you can convert the interruption into total disruption by using a variable you need as the loop index for printout. Pick a variable the program doesn't know for that job, and don't forget to complete the loop to restore the stack.

The NEW command also ends the program and effectively erases it in the process. It seldom belongs in the program at all. The only case known is where you intend to punish the user. If a password is needed to access the program and the wrong one is entered repeatedly, you may want to NEW the program to force the user to start over. Especially for younger users, it's convenient to terminate the program with something like:

```
PRINT"Type RUN to play again when you're";:end
```

It will give a nice final display. For good measure, put in an appropriate GOTO. For example, after asking "Are you through?", use an END for a "yes" response, but follow it with the "no" action. Then the user who changes his mind can recover simply by entering CONT.

Of course, you may be ending one program only in order to start another. Remember that the LOAD command within a program implies a subsequent RUN but does not CLR variables. If the second program is no larger than the first, the old data will still be in memory and still be accessible unless you unlink them. (What the LOAD causes is not a RUN, but a jump to the start of BASIC—in effect, a GOTO the first line of the new program. If the load failed for any reason, it will cause reexecution of the old program without a CLR.) If the new program is larger than the old or has an early CLR, variables will be cancelled before it begins to execute.

Chaining programs by LOADing the second from the first was a frequently used device in smaller machines (e.g., 2001-8), especially when you could not assume a disk. On larger modern systems, it is better to put more into each program. If chaining is still needed, holding the linking data on disk is preferable to relying on their staying accessible in memory. For example, a bad program file could lose the work of its predecessors in the chain. Putting data on disk and executing a CLR when the next program begins is safer. An extra benefit is that bringing in a smaller program without a CLR does not release the memory between the end of the new program and the end of the old. A CLR before the LOAD or at the start of initialization makes all of memory accessible.

3.10 Coding tricks—DEF and ON

There are many features of BASIC which can be used as "tricks" to save code and effort. They are often relatively hard to learn and may be beyond the skill of the novice programmer. Often, that's just as well since they can be hard to understand from a listing. This section deals with two commands, DEF and ON, which are worth the trouble to learn and use.

DEF is a means by which you can extend BASIC using BASIC itself. It allows you to DEFine a unary numeric function which you can then call from any point in the program. In other words, you can replace a subroutine that computes one number from another with a user-defined function.

An example of the use of DEF is in converting a number to a character. The ASCII function does the job, but with two disadvantages: small numbers are unprintable characters, so are hard to verify, and some numbers correspond to delimiters which cannot be read with INPUT# commands. Many applications need to pack integers from 0 through, say, 200 into single characters to save storage. If we remember how to count in hex, we can extend the sequence: 0, 1, . . . , 9, A, B, C, . . . , chr$(255).

The algorithm is simply:

```
x$ = chr$(48 + x) if x < 10
x$ = chr$(55 + x) if x > 9
```

We cannot put an IF test into a DEF statement, but we can use another valuable "trick"—evaluation of Boolean expressions. Every logical expression is evaluated to "true" or "false," and those are given numeric values.

"True" corresponds to the number – 1, which is all ones in binary. "False" corresponds to 0, which is all zeroes. As a result, the rules of logic are preserved with simple arithmetic: a true statement OR anything is true (– 1), AND anything is that thing; a false statement OR anything is that thing, AND anything is false (0). But we can use the number that results from a Boolean (logical) expression as a number. So, we may write:

```
x$=chr$(48+x-7*(x>9))
```

The operation gives characters that can be INPUT or INPUT# just as though the IF test had been carried out explicitly. But now we can use DEF to generalize the job. In particular, we may define a packing function FNP and an unpacking function FNU by writing:

```
9010 DEFFNP(X)=48+X-7*(X>9):DEFFNU(X)=X-48+7*(X>64)
```

to get the desired unary functions. Now, you can verify that the functions are inverses of each other by cycling through fnu(fnp(i)) for i = 0 to 200. Notice that the argument for the unpack function is itself a function; as usual, the logic works from the innermost parentheses out. Do you want larger numbers than 200? Then repeat the function working in base 201—a little more complicated than base two or ten or sixteen, but the same in principle.

One advantage of user-defined functions is obvious—they can save program space. A second is that there is only one place to make a mistake; fix it there, and you're sure that it's fixed wherever it occurs. Of course, the user-defined function is most valuable when the expression is complex and used very frequently in your code, but get familiar with it in simple cases before you really need it.

There are two ON constructs that seem more difficult than they are: ON . . . GOTO and ON . . . GOSUB. Either replaces a family of IF . . . THEN lines and can save a lot of code. Since they're very similar, we'll look at ON . . . GOTO for our example.

Suppose we have a counter which indicates to which of four routines we want to branch. We could write:

```
110 IFJ=1GOTO1000
120 IFJ=2GOTO2000
130 IFJ=3GOTO3000
140 IFJ=4GOTO4000
150 GOTO100
```

All that is legal, proper, and wasteful. What it does is direct the program to 1000*j if j = 1, 2, 3, or 4; otherwise, it sends it back to 100. We can accomplish almost the same thing with:

```
110 ONJGOTO1000,2000,3000,4000:GOTO100
```

The differences are subtle but significant: a negative argument (j < 0) is an error, and fractions are truncated in ON. . . GOTO. Operationally, those limitations are seldom important, and you can save 73 bytes in this simple example (32 bytes instead of 105).

Another use of ON. . . GOTO is to provide a kind of IF. . . THEN . . . ELSE structure in a BASIC that doesn't have the real one. Suppose we want to use the yes/no utility (see 3.6 Keyboard input) to branch to 1000 (if "yes") or 2000 (if "no"). Then we might write:

```
110 GOSUB8050:ONJGOTO1000,2000
```

That's all it takes. It becomes even more valuable if the branching takes place under an IF . . . THEN test, where we can rewrite:

```
110 IFX<4GOTO150
120 GOSUB8050:IFJ=1GOTO1000
130 GOTO2000
150 ...
```

and, in the much simpler form:

```
110 IFX>=4THENGOSUB8050:ONJGOTO1000,2000
150 ...
```

We have not only saved a lot of program memory; we may also have built more readable code.

ON. . . GOSUB gives you the same features as ON. . . GOTO, with the added advantages that you save the THEN you would need on IF. . . THENGOSUB, and you save storage by replacing all the GOTO's after your destination addresses with RETURN's. The power of the ON is so great that you will often find yourself using it with only one value of an argument—where an IF would do the job about as well. In those cases, remember that ON. . . GOTO and ON. . . GOSUB will make your programs harder for a novice to read than simple IF's.

4 Devices

All Commodore computers use a simple, straightforward way of interfacing with the outside world. Anything that communicates with the CPU is a device, and talks with the computer by its device number. The keyboard is device 0, datasette is 1, and screen is 3. In machines with dual-cassette capability (PET, CBM), the second cassette is device 2; in other computers, device 2 is used for the RS-232C port. As default values, disks are assigned device 8 and printers device 4. Device numbers can be changed within many peripherals by hardware or software.

When the computer talks or listens to a device, it uses a constant "protocol" (set of rules) regardless of what is on the other end of the line. In general, you establish that communication by opening a file to the chosen device, then receiving data through input commands or sending data with print commands. Defaults are provided in the form of special commands for communicating with devices 0 and 3 (keyboard and screen).

In other words, PRINT is a special case of PRINT#, and INPUT and GET are simplified forms of INPUT# and GET#. When you use PRINT#n, you are telling the computer to send the output to logical file n, which had previously been opened to a numbered device. When you use PRINT, you are telling it to send that information in the same way to the default device, which is preset to 3 (screen). To change the default device, we use CMD. For example, CMD4 changes the device for PRINTing and LISTing to the one designated when you OPENed logical file 4. There are conditions in which you may want to open a file to a device which doesn't need one—such as the screen or keyboard. For example, suppose your output can go to screen, disk, or printer at user option. You could write separate routines to PRINT to the screen and PRINT# to the others, or you could write a single routine for all. To do that, just open a logical file to the selected device: 3 (screen), 4 (printer), or 8 (disk). Then, the routine that does the PRINT#'s and CLOSEs the file will do so to whatever device you are using. Using this feature will demonstrate graphically that PRINT and PRINT# are functionally the same operation; output to the screen will be independent of which command is used to talk to it.

On the PET and CBM, hitting RETURN in response to an INPUT prompt drops the system out of BASIC. In the newer machines, it leaves the value of the INPUT variable unchanged. In either case, a program might be better off with other results. One way to change the operation of INPUT on a null string is to use INPUT# instead. Open a file to the keyboard and INPUT# from that file. As with screen operations, the keyboard is a device, and file operations may be performed on it in any logical way. Of course, an attempt to print to the keyboard (or to input from the screen) can give you a file error ("not an input file" or "not an output file") if you opened it properly and accessed it wrong. And you can expect a long wait for input from the screen or for anything to show up that was PRINTed to the keyboard, but the system will do what you command, whether it is logical or not. Note that devices 0 through 3 are preassigned and usually cannot be reassigned.

When you send the same information to different devices (or receive it from different devices), there may be differences because of the nature of the device. That's a complicated way of saying that getting literals on a printer takes a different protocol from printing them to the screen—and that you can't print them to the disk at all. On the disk, CHR$(193) is just CHR$(193). On the printer, it is either a spade or a capital A, depending on what went before it. On the screen, it is either a spade or a capital A, depending on the value in one memory location—or, on the VIC or 64, depending on the values in an area of memory.

While the Commodore peripherals are smart, your program has to complement their intelligence by following the protocols they require. For example, if you want to see literals instead of graphics, you *must* use the appropriate POKE for the screen. You may either use SA7 or cursor-down on every line to the printer. The question is meaningless on disk or tape. The code that opens the file to the device should set up all conditions required for its protocol; thereafter, the routine that communicates neither knows nor cares which one you were using. For example, if you are sending output to a dot-matrix printer or the screen, you may define a paging string consisting of the CLR character and the HOME character. By defining it as CHR$(12) for the 8300, you may also run with a letter-quality printer. Preset the paging string when output is begun, and let the print routine use it without caring which it is. (It would be nice to use all three characters in the string and be independent of the choice of printer, but you can't. The 8023 printer will accept either paging command; given both, it puts out an extra page. And the 8300 prints CLR as "3"!)

Assignment of file numbers to devices is almost entirely at your discretion. However, some system is worthwhile if you are to be able to read your code easily. For example, you might choose always to input from even-numbered files, always to output to those with odd numbers. Or it may be convenient to use file 1 for a primary file, 2 for secondary, and so on, regardless of the file's function or the device on the other end. If you do that, it is convenient to CLOSE1 routinely when entering the menu or at any other major break point in your code. It can do no harm (no error results from closing a file that isn't open), and it can correct a problem if you got to that point unexpectedly. For example, if you broke out of BASIC inadvertently and did a GOTO the menu, or you took an error exit, you want to make sure that whatever files were open to disk are closed before operations continue. Advanced software usually requires use of many files on many devices, and some ordering scheme for your own convenience will help you keep track of them.

The one restriction on file numbers is that the Commodore system provides an automatic line feed (chr$(10)) if you assign a file number greater than 128. On Commodore peripherals, the line feed is either unnecessary or wrong; keep file numbers under 128 unless you're using someone else's device which requires an explicit line feed from the computer.

In opening files, secondary addresses can be crucial. Some SA's are consistent among devices of one type (e.g., disk drives); others vary depending on the particular model in use. Where interoperability is important, use SA's and features common to all models that are likely to be used. For example, even Commodore printers vary in their formatting and special characters; instead of using SA5 for formatting, you may build a print string as you need it and your code will run on all printers. In practice, many of the secondary-address functions are inconsistent on the printers, including variable line spacing, literals/graphics mode selection, and formatting. Although it puts code into the computer that belongs in the peripheral, you are better off not relying on the printer to do all the things its intelligence is supposed to allow.

4.1 Un-conventions

To bring order out of computer chaos, international conventions have been established. They represent protocols accepted over many years, some dating from the most primitive teletype machines, and are accepted by all manufacturers of computer hardware and peripherals. Except Commodore.

When Hewlett-Packard started making computers, they wanted an efficient, high-speed interface with their peripherals. They devised a buss system which came to be known as HPIB. Other manufacturers developed peripherals to work with the HPIB, and to reduce publicity for Hewlett-Packard they called it the GPIB. The Institute of Electrical and Electronic Engineers embodied the GPIB in its standard, IEEE 488. Key properties of the buss include its requirement for intelligence on both ends (to manage the protocol), its high speed (suitable for even hard disks), and its implementation by many vendors.

Given the low cost of microprocessors, Commodore was wise in selecting the GPIB for its computers. Unfortunately, they chose to economize by omitting some parts of the protocol that were not needed in the earliest machines. As a result, Commodore peripherals will not simply plug into IEEE 488 busses, and Commodore computers will not run all GPIB peripherals. Correction of the problem of interoperability is well documented, and requires both hardware and software—for you to buy and to build, respectively.

The situation with the RS-232C serial buss is somewhat different. In the VIC 20, Commodore has implemented all of the required logic, but saved substantial cost by not putting in the extra power supply for its plus-and-minus twelve volts. Since the adapter to change signal level can be as simple as an inverter, they provided inverted logic as well. The 64 behaves the same way, except for its inherent timing problem. The Commodore 64 looks into memory before refreshing the screen. It must do that in order to handle relocation of screen memory. Unfortunately, stealing that glance takes CPU time that is given the highest priority in the interrupt structure. As a result, clocks are given lower priority and do not run with their customary regularity. That affects all busses in the system. The intelligence of an IEEE interface unit can handle the problem, but the serial busses have more difficulty. Since the RS-232C buss demands precise timing, designers are hard at work trying to build interface hardware that will work consistently in the face of the 64's inconsistency. (That same problem may be part of the reason for difficulties with the serial peripherals from Commodore.)

One standard that no one but Commodore violates is ASCII—the code converting between characters and numbers. In computer terms, it's ancient, venerable, and respected by (almost) all. Commodore's logic for changing it stemmed from their incorporation of literals mode in all machines. Since they expected users to want upper- and lowercase characters in their programs, they recognized that it was easier to use the keyboard as a typewriter.

In ASCII, the unshifted characters are uppercase; in Commodore, they are lowercase. The ASCII standard is not complete, and there are variations in its special characters among manufacturers. If your external devices want true ASCII, you must convert to their requirements from those that Commodore supplies. In some cases, you may simply map characters 65–90 into 193–218 and vice versa. The logic is a little easier if you mimic the printers by putting out the wrong case for characters 91–95 too, but in either case all you need to do is XOR the character from the computer with 128 to get the ASCII value. If your peripherals need special characters, check their translation against Commodore's documentation and implement what you need.

Whatever reasons Commodore has had for varying from international standards, your software and systems may have to correct for their results. Commodore stays consistent within its own lines and really hasn't been concerned with whether you have problems tying in to others'. Other manufacturers use other approaches. For example, Apple expected you to plug in boards and chips from other vendors for what you need, and even to use an outside source for lowercase characters. No one fully implements all the protocols for all the different lines of printers, but some manufacturers come closer than Commodore. Like Apple, Commodore has chosen a path that supports a substantial cottage industry in peripheral adapters. Unfortunately, each such adapter is likely to require its own software. Interoperability of your programs will suffer from that variability. One routine you are almost certain to need for telecommunications is ASCII conversion for upper- and lowercase letters. Otherwise, plan to adapt your software to each installation, with its unique hardware and interfaces.

4.2 Printers

Commodore distributes a wide range of printers manufactured by other companies. The two designed for the serial buss of the VIC and the 64 are the 1515 and the 1525; they differ in paper size (the 1525 takes standard width), in noise level, and in that the 1515 will not run correctly with the 64.

The range of printers on the IEEE buss is substantial. The original 2022 and 2023 required an upgrade ROM for best performance. Commodore provided the ROM without charge, and embedded only one time bomb for the programmer: paging logic was changed without warning and without documentation. The later dot-matrix printers are quieter, faster, and more costly. Their firmware is superior in that they more reliably perform the functions associated with secondary addresses than the earlier machines could manage. The 8300 letter-quality machine is a Diablo printer interfaced with a CMC adapter. Both are competent, but the adapter requires support for its edge connector. The combination is effective, professional, and demanding of code different in many ways from that for the dot-matrix machines. Typical of the changes required for the 8300 are the modifications discussed in Section 5.7 SUPERLISTing.

The dot-matrix printers carry dual character sets, approximating Commodore graphics and literals. Since the printers have either six or seven dots per column and the screen has eight, the graphics are not exact. Whether or not you care for the seven-wire literals, the six-wire version in the 1515 and 1525 has no descenders and cannot produce quality copy. All dot-matrix printers shift character sets reliably with the cursor-down character (literals) and cursor-up (graphics). Most will usually shift into literals mode (inverting operation of the cursor up/down) when a file with SA7 is opened, printed to, and closed. Some will shift back to graphics with SA8; however, SA8 sometimes works and sometimes does not, depending on the particular printer, how often and how loudly you send the command, and the temperament of the little gremlin who pushes the wires.

The five characters (91–95) immediately above the alphabet behave strangely in all dot-matrix machines except the 1515 and 1525. In literals mode, square brackets, up and left arrows, and backslash (English pound sign) print in reverse case. On the 1515 and 1525, cases are correct. On the 8300, characters depend on your print wheel, but there will be fewer than the dot-matrix machines can generate. The standard underscore character is the keyboard left arrow. Underscore on the dot-matrix is a shifted dollar sign, which prints as dollar sign on the 8300.

On any printer except the 1515 or 1525, carriage return is possible without linefeed, permitting double printing on the line (e.g., for underscore). On the 8300, an ESCape sequence is required; on the other printers, chr$(141) does the job. Paging on the dot-matrix machine uses the HOME and CLR characters. On the 8300, chr$(12) is required, while the 4022 or 8023 responds to either. For safety, flexibility and interoperability among machines, it is worth counting lines of text and paging by sending chr$(13)'s or chr$(10)'s. Similarly, formatting within printers is quite variable and should be relied on only when you are certain that, say, an 8023 will be used for all printing from your program. Otherwise, form your own print strings within the computer, and do not tax the printer's intelligence with SA5's and the like. (One feature that works consistently on dot-matrix machines is the user-defined character. It even seems to behave the same regardless of the IEEE printer used, a truly remarkable feat of consistency for Commodore.)

In many respects, the slow, noisy 2022 is the preferred dot-matrix printer. It is reliable and well built, has a clear typeface, uses standard teletype ribbons (six-hole variety) and doesn't consume them too rapidly. The cartridges in other printers are more costly and run out sooner, hang-ing up the system. That property is important for unattended printing. In the same vein, remember that the 8300 lacks a paper sensor; if it runs out in the middle of a long output, you can damage the platen—as well as waste time. With that caveat, the 8300 is the only choice of printer for professional use.

A wide variety of non-Commodore printers can be interfaced with any Commodore computer. An RS-232 port is a good choice when a suitable interface is included. A letter-quality printer can then be attached for less than $1000, a considerable saving over the 8300. Other printers can be run from the user port, requiring only a connector and appropriate software. The software is the problem in those cases; you will probably have to write your own, and any resemblance between talking to the printer as device 4 on the standard buss and talking to it on the user port is coincidental. You will have to correct Commodore ASCII to what the rest of the world uses, and handle paging and other features in detail. If there is any possibility that you will want to use such a printer, collecting all print commands into a common subroutine becomes mandatory. You can then develop and verify your code on a standard printer and make the changes later in one place for all output.

You will minimize the cost of changing printers if you collocate all code in a restricted area of the program. If your system is to run with multiple printers, you may want to make selecting among them a removable line in initialization, or a question asked each time the program runs. If the user is likely to have only one printer, a run-time option to remove a line is usually preferable. If the system will have multiple printers at one time, ask the question during initialization.

4.3 Disks and drives

All Commodore disk drives require high performance from the disks themselves. You will need to find the best disks consistently available, and even then to check out each batch that you buy. Even the best manufacturers have had periods of poor quality control. If a few bad disks slip into your system, vital data can be lost despite your backup procedures.

You and your customer will profit from documented procedures for handling disks. Protection from dust and fingerprints is vital, so set up a system for keeping the disks clean and protected. It is worth the effort to format each disk when the box is first opened. Gross faults will be detectable by appearance, difficulty in inserting the disk into the drive, and the sound of the drive in operation.

Most faulty disks will fail to format, so they can be discarded (or returned) before any information is committed to them. If your program has to format the disk, it will take several minutes at each initialization. (You can detect whether the disk is already formatted and avoid wasting the time. The easy way is to try to open the directory to read. If you succeed, you need not reformat the disk. That step will also detect the wrong format, except that it won't catch a 2040 disk as an error; to do that, you'll have to read the format code at the end of the first directory line. Since the 2040 format is obsolete, you may be able to ignore the problem.)

Considering their price and the quality of the competition, Commodore's drives are quite good. The single drives for both IEEE and serial busses are generally less reliable and less sturdily built than the duals. The IEEE version performs well when properly set up, but its setup is difficult and it has a disturbing tendency to lose adjustment. Upgrade boards are technically available, but whether they cure the problems is not yet proved. The serial buss machines have firmware problems that have not been isolated at this writing. The 4040 dual drive is highly reliable and not too demanding on the disk itself. The 8050 comes from one of two manufacturers. The one with a 4040-style door is relatively slow, but has proved reliable; the one with a 1541-type door is quite fast but has had a painful history of failure. Both 8050's use higher density than the 4040, so demand even more of the disk itself. The 8250 double-sided drive offers little advantage for its higher cost; its performance is much the same as that of the 8050. The problem with the 8250 is that its cost approaches that of a hard disk, while its performance and capacity are still in the range of the floppies. There may be applications where the 8250 is *the* right answer, but more often you will either need to use a hard disk or be able to get by with the readily available, better-known 8050 or 4040.

Commodore's hard disks are just becoming available as this text is being written. They should be given at least six months in the field for debugging before you design advanced software around them. They are to be fully compatible with the logic of the floppies (which is common to all current units), so you should be able to design your programs for an 8050 and upgrade to hard disk without change.

The major advantage of hard disk is not its increased storage—although that is significant to many applications. The big payoff is in speed of access to information. The disk spins faster and has higher bit density than a floppy, so data are pumped in more rapidly. In addition, the time to find the data is decreased by the head arrangement. The hard disk is fast enough to allow the luxury of storing a directory to your files on the disk itself; you can make an indirect access to information faster on the hard disk than you can a direct one on a floppy. (And if you aren't accessing the disk continuously, you get a side benefit since the hard disk doesn't have to spin up to speed before occasional reads.) While your floppy-based program will run better and faster on the hard disk, you won't get all of its advantages until you design your programs to exploit it.

4.4 Disk files

Advanced software relies on disk storage to supplement that inside the computer. Tape can hold only sequential files and programs, and operates so slowly that it is used for no serious work. Much of the discussion of sequential files in the following can be applied to tape, but to little purpose.

The disk system supports four types of file: program, sequential, relative, and user. The user file demands direct control by the programmer; the level of skill required to use it is too great for the purposes of this text.

A program file consists of an image on disk of the contents of memory SAVEd by a program. It carries the address from which the recording began (for a BASIC program, start of BASIC) in the first bytes, then a byte-for-byte copy until the end (for BASIC, the occurrence of two consecutive zeroes). Disk channel 0 is reserved for reading the directory or loading programs. Channel 1 writes the directory or saves a program. The disk directory is recorded as a kind of program, with the special names "$" (both drives), "$0" and "$1." To open the directory or any program file for reading, OPEN1,8,0, "fname." Since the characters in programs include delimiters, you will have to GET# to read the information, but your program can access the individual bytes of the program or directory in this manner.

A similar operation using channel 1 allows you to write a program directly (rather than saving it), but this capability is of relatively little use. By reading and writing programs (rather than LOADing and SAVing) you can copy material which cannot be duplicated otherwise. For example, a simple BASIC program can transcribe the usable parts of a program which has been damaged by a disk fault.

A sequential file contains the sequence of characters sent to it in PRINT# commands, including automatic carriage returns unless they are suppressed with the semicolon. If there are individual records in the file, they occur because you put them there. Since there is no fixed spacing of records in the file, no storage is wasted with blanks to fill the space to the next record. That's efficient storage. Unfortunately, it comes at a high price: you can't get to the middle of the file except by reading through from the front. The disk operating system (DOS) has no way to find any point in the file except its beginning and its end. The only interesting things that a program can do to a sequential file are read from its start and write after its end.

Suppose you have a list of 100 names in a sequential file delimited with carriage returns. All you want to do is replace #37, "Hector," with "Hecuba." The straightforward way is to open a dummy file for write, then to INPUT# from the old and PRINT# to the new 36 times, then PRINT# "Hecuba," INPUT# (and discard) "Hector," then INPUT# and PRINT# until the file is empty. Finally, close both files, scratch the old one, and rename the dummy with the old name.

This process is effective but slow and clumsy. It also has some traps in it. Run on a single drive, it will cause the head to search back and forth repeatedly, shifting between the file it's reading and the one it's writing. You will quickly learn to recognize the sound. And don't forget that both scratching and renaming are risky operations; each should be followed by initialization, and a collect (validate) would not be amiss.

The example gives insight into when you would *not* use a sequential file. You should use it (and save storage) when you will use the file as a whole, rather than piece by piece. Two cases arise frequently: a memory-resident file and one accessed by content. In many cases, the file must be manipulated substantially during a run of the program, or it must be sorted in multiple ways, or accessed so frequently that you can't afford the time to keep going out to disk. Then you load the whole file into memory to begin operations, manipulate it as required, and finally replace the file as a whole if any changes have been made. For that sort of handling, a sequential file serves quite well. Similarly, you may have to leaf through a whole file every time you use it simply because you never know which record(s) have the information you want. If all accesses are based on the values of the records (their content), rather than their position, a sequential file will do the job well. Finally, you sometimes just want to use a file to sequence the operations of a program; in that case, again, the sequential type may be just what you need.

A relative file is a collection of fixed-length records for which the DOS has the means of accessing by record number. To hold the same amount of information as a sequential file, a relative file needs more disk. One part of the space lost (overhead) is due to fixing record length. The size must be large enough to hold your longest record (any excess is simply chopped off). Therefore, you may need a 12-byte record size, even though you will only use an average of four bytes per entry. The remaining two-thirds of the space is simply lost. In addition to empty space, the disk must also hold the information on where the records are located. The pointers to the data are held in "side chains," which also take space. They also take an extra channel of the disk drive. On a dual drive, you can have two relative files open at a time, and even have room left for a sequential file. On a single drive, only one relative file at a time, please.

There are times when a relative file is the obvious (or only) choice.

Ideally, it is used for records which are all of constant (or nearly constant) length, and which are to be accessed by ordinal. They are perfect complements to sequential files, since they are ideal when they are very large and require access to individual records for read and write. Unfortunately, it often happens that the file you need fits neither extreme; it may be of variable-size records which need individual replacement. Then you must choose between the evils, and spend either a lot of disk or a lot of time using either relative or sequential types, respectively.

A little creativity can save a lot of grief. Suppose that you need to keep a lot of information about a number of people. We might use a sequential file of names, reading it in (and writing it out if necessary) as a whole. The order of the name in the file (its ordinal) is the number of the relative record of information about that person. Suppose the people are students and that we need a full school year's record for each one, with one character for each of 200 days. The longest name may be 30 bytes, but they may average 14. So the memory-resident file for 500 students is only about 7K. The disk-resident data file is 100,000 characters long, but that's okay; we only look at one student at a time, so we only need 200 bytes of memory. We can manipulate the names in memory, then fetch the data record we need, manipulate it, and put it back where it belongs. The only time we would have to write the name file out again is when we have changed it—a rare event for this type of application.

Note that we can work around many of the limitations of the file types by combining their virtues and cancelling their faults. Sequential and relative files are complementary, and most advanced software will need both. Where the trade-off in algorithms is usually as simple as speed versus memory, in disk files it is usually among speed, computer memory, and disk memory. And even the speed question is complex since there are designs that take more time to initialize and less to run than alternatives. (A simple instance is between reading the whole file in once and manipulating it in memory, compared with modifying it on the disk. Each modification will take multiple disk accesses and a lot of time compared with shuffling memory, but you don't have the long delay to load the file in the first time. There is no general rule for designing the file structure. Pick the architecture that fits your application.)

4.5 Disk commands

There are two ways to send commands to any Commodore drive: through the command channel or through BASIC 4.0. In fact, the disk receives the same command, regardless of the route you take to send it. As a result, you can use relative files, append to sequential ones, or perform any of the other BASIC 4.0 operations on a VIC or a 64. Sometimes it will take extra code, but seldom enough to matter. And in many cases using 2.0 commands is easier, more logical, or simply more effective than using those of 4.0.

The most useful information you can get about the status of a disk is in its directory. When all files have been written properly, the sum of the sizes of all files added to blocks free is the total number of blocks on the disk (664 in the 2a format of a 1540, 1541, or 4040). Many disk operations lead to slight confusion in the directory, so the block count increases by one to three. Attaching information (APPEND or CONCAT) tends to create this anomaly. However, when the error exceeds about five blocks, the directory may have become seriously confused. At that point, files may go into hiding—perhaps irretrievably. Repeated APPENDs to a single file do not seem to create the problem, and no consistency has been found in losing files, but it happens with painful frequency. The first file on the disk has never been lost in this process, so it makes sense to put really critical material in a file on a NEWed (HEADERed) disk.

There appears to be some risk on other commands that modify the directory. For safety, reinitialize the disk after each SCRATCH or RENAME; SAVE@ should be avoided since you cannot initialize between the scratch and the save. An occasional COLLECT command only hurts in that it spends time; it will tend to reduce the extra blocks that the DOS invents for you, although the only way to eliminate them is to COPY the files to a blank disk.

The most common form of catastrophic directory failure (even on a 4040 or 8050) is that two filenames point to the same physical file on the disk. On the 1540/1541, a more severe problem can occur. Reading is a difficult process for people and for disk drives; given enough material, either can get tired and confused. When a 1541 reads too long, its firmware simply quits. Depending on the type of disk used, the ambient temperature, and the phase of the moon, reading a hundred blocks or less on a 1541 may give you a "disk id mismatch." Since that error cannot be real, it simply tells you that the DOS is lost. On consecutive runs with the same disk and program, disaster will strike in different places. There is no solution except to keep your files small and to avoid long periods of reading in files from a 1540 or 1541. Those drives are suitable for programs, moderate sequential files, and limited-use relative files; they cannot be counted on for maintaining critical records.

Commodore disk commands can be used to give you a reliable system for even critical data if you take care to maintain backup copies and if you integrate recovery procedures into your products. On a dual-drive system, HEADER one drive at the start of the day, open a Log for write, then CLOSE it. When activity is to be recorded, APPEND the transaction, then CLOSE the Log. At the end of the day, merge the Log into the rest of the data system. Since the Log is the first file on a blank disk, it is likely to survive any anomalous disk behavior. Your backup procedures should allow repetition of the merge process; any failure is unlikely to repeat. The disk with historical data should not be written at all until successful merger has been verified. Verification may be as complex as checking the block count by reading the directory; if you're going to that much trouble, take the time for a COLLECT first.

Except for a bug, the BASIC 4.0 command HEADER is equivalent to sending a NEW to the command channel. The undocumented error is that a variable cannot be used for the disk id in HEADER; if you are going to format the disk within your program, you must revert to BASIC 2.0. One factor to consider in promoting both program speed and drive reliability is head movement. When one drive is accessing two files concurrently, the head must be moved between them. That takes substantial time and wears the mechanism. If you are running on a dual drive, COPY files between sides, CONCAT from one side to the other, and generally avoid performing concurrent operations (e.g., read and write) on one drive when you can use both.

Secure processing on disk is tedious, but the user may not be troubled by the process. Merging is the last operation of the day, and requires no manual input when it succeeds. Even if the process takes hours to complete, it can be effected by running the system overnight. On the rare occasions when it fails, recovery can be the first job in the morning. Since computer time is essentially free, good program design will yield a successful product even when disk performance is marginal.

Directory confusion may be eliminated thanks to a suggestion by Mike Louder. The method is to validate the disk (COLLECT in 4.0) *before* any scratch, rename, or other erasure from the directory. The drawback of validation is that it is very slow, taking several minutes on a full disk. However, you may be forced to take the time penalty if your code has trouble with disk errors. Since the Commodore drives have enough intelligence to validate without computer intervention, you may send the command early and have it implemented in the drive while the computer is doing useful work. For example, a program which reads in a file for modification expects to scratch and rewrite it later. Therefore, you may want to COLLECT the disk it came from after reading the file, rather than waiting until it's time to scratch it.

4.6 Tokens

A token is a single item that replaces a collection of others—in the way that a subway token replaces a handful of change. In the BASIC interpreter, the nine characters of DIRECTORY are replaced by chr$(218). Interpreted as ASCII, that token would be a "Z," but as a BASIC 4.0 command it means DIRECTORY. The reason that the shorthand form "diR" works is that finding a capital letter causes the interpreter to seek a suitable token for the part of the command already input. If you had entered "dI," the list of commands would be searched, and the first one that fit, DIM, chr$(134), would be selected.

A data-intensive program may also benefit from tokenizing under your control. Suppose you use a system for converting the 200 most common entries in a field into single-character tokens. One way is with the numeric packing and unpacking functions (see 3.5 Bits, bytes, characters, and numbers) that create characters you can INPUT. If you have ten thousand classical recordings to file, the 200 composers most commonly found probably account for 9,990 of them. To handle the other 10, save one token to mean: this one is spelled out. If the average composer's name takes seven characters, you replace seven with one in 9990 cases, seven with eight in the other 10. Your primary file's composer information then shrinks from 70,000 characters to 10,080. However, you need a secondary file, a "directory," of the names to which the tokens point; that one is 1400 characters long (7*200). Since it is a separate file, its size may not be a problem; in any event, the net saving in storage is about a factor of six.

A little more storage is saved in a real system. The composer field of the untokenized file requires a character to identify its end, a "delimiter." It marks the end of a field which may be anywhere from two to, say, twenty characters long. Considering the delimiter, the file would then have 90,000 characters. But for the 9,990 items that are tokenized, the field is fixed at one character, and needs no delimiter; only the 10 exceptions need to specify their length. Then tokenizing reduces that field's size from 90,000 to 10,090, or total storage to 11,490.

Accessing the tokenized file requires packing, unpacking, and conversion between numbers and characters. Therefore, using it will take longer than using the simple one. As usual, saving memory costs time. Since disk access is usually slower than computer operations, you can limit the slowdown to acceptable levels. Count on thinking about your design for several extra hours if you're going to tokenize effectively.

One thing you must resolve at the start of planning a tokenized file is where to keep the directory. For maximum savings, it can be held as a sequential file on disk and read into memory at initialization. That will take a chunk of precious RAM. However, using the file will be a lot faster than if you access the directory on disk. If you simply can't afford to spend the RAM, then try making the directory a relative file. With a floppy, the time for the extra access can become painful; your program will have to avoid reading any directory entries it can, and still may not be fast enough. Of course, if you're using a hard disk you have enough speed to keep the directory out there. One solution is to design the system with the directory on disk, build it with the floppy, then invest in the hard disk after everything is running as you wish. On the other hand, with all the extra storage on the hard disk you may not need the space-saving that tokens offer.

Tokenizing a hard disk file pays off in other ways, depending on how you plan to extract records. For rapid access to information, you will need pointers for many fields to lead you from record to record. To get that capability, you will need a directory for each chained file. Tokenizing them costs little and saves much. (Notice that in many ways a tokenized field is easier to maintain. For example, spelling can be modified in one place for all records. Without tokens, each record with the old spelling would have to be found, corrected, and replaced.)

4.7 Using files

A real example of the use of advanced programming techniques in managing files would be both too big for this text and of too little general value. Instead, we will work with an artificial case that collects many of the ideas in a way you can transfer to your own programs.

Suppose our problem is to translate a telephone area code into the state or province in which it is located. The longest name we need is 14 characters. If they averaged 8 characters and we allowed for codes from 100 through 999, we could hold them in 7200 bytes of memory plus overhead. Logically, that large a file should be out on the disk. For speed, we'd use a relative file whose index was the area code and whose entry was the state name. For that, we'd need 900 records of 14 characters, totalling 12,600 bytes plus overhead (side chains, directory). Given all the space on the disk, that method should do the job. But we can do much better with a little effort.

The first step is to look at area codes themselves. All have either 0 or 1 for the second digit, for reasons built into the switching system. If we swap first and second digits, making 213 into 123, we have a scheme in which the 900 area codes compact into less than 200 (the magic number for the packing function we've used before). We could use a subroutine to do the job:

```
10 X$=LEFT$("0"+MID$(STR$(X),2),3)
12 X$=MID$(X$,2,1)+LEFT$(X$,1)+RIGHT$(X$,1)
15 X=VAL(X$):RETURN
```

(Line 10 is complicated because a leading zero is sometimes required.) That routine transposes the first and second digits of a three-digit number, which is what we had in mind. And the routine is its own reciprocal —since it just swaps two digits, it recovers the original by swapping them back again. Just by using the swapping function, we cut storage from 900 records to 200.

The next thing to look at is the names of the entries themselves. On average, there are about 3 area codes per state. Therefore, we store each state name three times. There are about 60 states, provinces, and other areas to which codes are assigned. Suppose we build a file of those names, hold it in memory, and then keep in our file of area codes only the number of that entry. When we initialize the program, we read the state names into an array from either DATA statements in the program or a sequential file on the disk. (We could use a second relative file on the disk, but that would slow things down. On a 1541 drive, we would have to keep opening and closing the two files, which would be particularly slow and would stress the drive.) We now save the 60 names once, then have 200 pointers to them in the relative file. If we kept those pointers on disk, we might pack them with fnp and fnu, giving us a one-character record; in memory, we might prefer to use them as integers (two bytes each) and save the packing and unpacking time. Depending on the application, we'll be dealing with about 700 bytes of storage instead of 7-12K.

There is yet a final step we could use in packing the data into the system. We could put the pointers (the number of the state or province in our list) into a single-character string. Since the string is less than 256 characters long, it is legal. However, some of its characters cannot simply be typed from the keyboard, so the string may have to be pieced together.

By now, the logic has gotten a lot more complicated, timing is a little slower, and storage is a lot less. In the system with a 900-record relative file, the logic was:

```
input the area code
send the code as the RECORD#
INPUT# the name
```

If we packed it as much as possible, we would first read in the file of names and then use the following for each access:

```
input the area code
swap digits
find that record with RECORD# or MID$
retrieve the character
convert the character with ASC and fnu
use the converted number as the index to the name file
```

The mapping, converting and user-defined functions are complicated. If you have plenty of memory and not much time, they're not worth the trouble. But in many cases, the time to input the three digits is likely to be longer than all that logic, so it won't be noticed. And there is a sort of law in programming that the problem will always expand to fill memory, so it is usually worth at least some of your effort to save space. Even in this simple problem, we have taken a program that would require a disk or at least 16K of computer and rewritten it to run in an unexpanded VIC 20 from cassette. At the other extreme, if we had a hard disk, we wouldn't worry so much about storage space, access time, or opening several relative files. If we were trying this simple problem on such a big system, brute force would be a logical way to do the job.

4.8 Sorts and searches

Among the most common functions of advanced software are the storage and retrieval of information. Those two operations are logical inverses of each other. There are commercial file managers available to assist in the tasks, but you will need to know how to design your own for many applications.

The basic question to be asked in designing data files is whether they are to be searched or sorted. In a sorted file, selected fields are designated for sorting. A list is made for each field, running from the first to last record by that field's ordering scheme (e.g., numeric or alphabetic). Each entry in that list is called a "key." It points to a record in the main list. When a record is added, its position in each list is found and its record number is inserted in sequence in each place. To drop a record, its number is deleted from each list. For a system with many small fields, those lists can be larger than the file itself; more often, they are of about the same size. Of course, to use the record number, the file type must be relative, which takes still more space.

A file which is searched for entries is similar to a card catalogue without cross references. Usually, the file is ordered by one field—say, alphabetically by last name. In order to find entries in another field, the whole file must be searched by reading it from storage. That can take substantial time for a large catalogue. There is no storage spent on lists of fields, and the file can be sequential (saving still more space). Typically, updates to the catalogue are merged into the file by copying it from one disk to another in a dual drive. The burden on the drive is not severe since the files are simply read in sequence; the head has little or no searching to do. Still, the process is slow. For a large file (say, 10,000 records) it makes sense to wait until there are many entries to be added or deleted (perhaps 50 or 100). The time used to add one record is about the same as that for a hundred since most of it is spent just feeding records from one disk through the CPU to the other. Merging into a large catalogue is the sort of job one gives the computer to run overnight.

It is necessary to be able to search any file if you ever need to look inside a field. To retrieve a record by anything on which you have not sorted, you must search the whole file—whether it is sorted on other fields or not. The less you know about how a file will be used, the less likely you are to set it up correctly. Then you will have to build it all over again when you know what sorting your customer really needs. Make sure that your file package will support special searches. The odds are good that you will need one someday on even the most completely sorted file.

Terminology in data management is more confused than in any other aspect of microcomputing. Consider a collection of data files and their associated lists (files) of sort "keys." It is reasonable to consider a "file manager" to be a package of code which supports the maintenance of a single data file and its correlated lists. Whether it is highly sophisticated or simplistic, easy to use or demanding on the programmer, as long as it works one data file at a time, it is a file manager.

A data base is a collection of data files with their individual and collective lists. A "data base manager" will keep the multiple files within the data base coordinated with each other and keep their lists linked as required. Whatever the package of commercial software may be, the chance that it is a data base manager is small. Software to coordinate multiple data files is complex and demands more of the computer and disk systems than most micros can manage. Even as simple a task as correlating a mailing-list file with data on the accounts it contains is beyond the capability of the commercial packages examined to date.

Sort keys may be kept on disk or in memory. On disk, access is slow but storage is cheap. In memory, you need RAM to save time. Note that ordering may be on a hierarchy of fields. If the primary field has multiple entries with the same value, they may be ordered by a secondary field, then by a third, fourth, etc.

If a file has the following characteristics, you should sort it.

It is to be accessed by several fields.
You know from the beginning how it will be used.
It will fit comfortably on the disk, even with:
  relative field overhead;
  sort-key overhead.
It is to be updated on line.

How the file is to be updated is a key to designing it right. A sorted file is best written one record at a time. The operator will intersperse entries with retrievals, and will tend the system as it does its work. Each addition to a sorted file takes many key accesses and significant time—usually enough to be noticed, but not enough to take a coffee break. If there are fifty updates at a time, the sort system is hard on the operator. A catalogue is updated in batch, with plenty of time for coffee—or lunch. But the operator need not watch the computer; all entries are made at once and the machine will digest them as a group. So part of the choice of system should be based on how it will be run.

The most important single factor in choosing the file structure is the software. For a catalogue, you must build your own, start to finish. That will take time, but give you complete control. For a sorted structure, you can start from any commercial package. Your selection among them should be based on your ability to interface the programs you are writing with the files it supports. In practical terms, your program(s) must be able to pull information from the files that their software maintains. Ideally, the file manager should come with routines that tie your program to its files and services. If it doesn't have the routines, you'll have to invent them. Since the package seldom comes with enough information to do the job easily, you may spend more time tying in to a purchased product than you would building your own.

Before setting out to build a file manager, plan on spending substantial time learning the techniques. Commodore's Name Machine uses a very simple catalogue ordered by a single field. After you understand how it works, you might modify it for a secondary sort key of first name. Then two entries with the same last name will be alphabetized by first name. The next step is to discard cassette capability and to convert the file the program now manipulates in memory into a relative file on disk with a single ordering key. If you get through that stage, try a second key with ZIP code first (that's necessary for the large files that a business requires). After those exercises, you should be ready to design and build your very own file manager. If you did a good job, that last exercise may be marketable in its own right, especially if you tie it in to other useful software the way that the Word and Name Machines are linked.

5 SUPERLIST—an Example & a Tool

A printed listing of a program is a necessary tool in understanding it. You know how to print such a listing, and know how much it means for both development and analysis of the code. But a conventional listing lacks some features that would help even more. If you had your choice of features for it, you might ask for these:

1) Spell out special characters.
2) Cross reference line numbers by those that call them.
3) Cross reference variables by line references.
4) Print a code number to check different versions.

The first item on the wish list is clear enough: those who use Commodore listings all the time know what reverse q's and s's mean. But they are hard to read from a dot-matrix printer, require reverse printing (impossible on letter-quality), and communicate little or nothing to those unfamiliar with the specialized jargon. So, let's convert reverse q into something like c-d, reverse e into wht, and so on. In the process, we'll get a listing that can be put out on a letter-quality printer. In a word, it will be suitable for publication.

The second wish is a little less obvious. When you know who calls a line, you get several useful data. First, if you want to discard a line, you had better be sure that nothing is trying to use it (GOSUB or GOTO). Next, if there are many calls to a line, you may want to make it conspicuous—say by numbering it ×500. Third, routines that are used a lot are worth extra effort to optimize. When memory is very tight, you may even want to save some by moving a small routine that's called often into the two-digit area; it will save a couple of bytes each call. But the most important use is when the program goes wrong. What you often need to know is: how did it ever get here? The line-number cross reference is often the only way to answer that question.

Variables need cross reference even more than line numbers. Do you use a variable thirty times? Then make sure it's only one character long. What working variables are available to add a feature in a routine? Check the cross reference. And when your program goes sour, you can figure out what code could have clobbered your pointer by looking at all of its references.

The jargon for a number which characterizes a program is "checksum." Its essential property is that two programs with identical checksums are very likely to be the same. To say almost the same thing in another way, it is unlikely that a change to the program will leave the number the same. Then assume that you have a simple program which computes the checksum. Run it against two programs to find out if they are (probably) the same. The more complex the checksum, the less likely it is that it will miss a program change. On the other hand, the more complex it is, the slower it runs and the less likely it is to be used. A practical compromise for microcomputer software is a simple, exclusive OR of the bytes—as long as sabotage is not suspected. Nothing less than byte-for-byte comparison will guarantee that two programs are identical, but barring a deliberate effort to make undetectable changes, eight bits should do the job well.

The need for SUPERLIST became apparent as soon as advanced software was attempted in BASIC. Most of its functions are handled by the compiler for FORTRAN or COBOL; they are much more difficult in languages such as FORTH. In the late 70's, the need for SUPERLIST emerged when coding the 8K PET. The version presented here has been recoded completely to include BASIC 4.0 and color commands. It illustrates many of the features of advanced software as developed in the earlier chapters. The program is both an example of advanced software and a tool to help you develop advanced software for yourself. One way to demonstrate what it does is in Section 5.7 SUPERLISTing—it has been run on itself.

5.1 SUPERLIST initialization

Entry to the program is at the first line (10); requiring the user to RUN nnnn is poor human engineering. So, line 10 carries GOTO9000 (the start of initialization) and the copyright notice.

Line 9000, like all major entry points, contains only the remark describing its function—initialization. The next line sets the parameters which the rest of the program uses: 10 (lines of text per page), lp (total lines per page), r$ (chr$(13)), a convenient base number (bs = 2 ↑ 15 − 1), and a user-defined function. The function is called "fnx" and provides an exclusive OR (XOR), a function omitted from BASIC but needed for our checksum. Note that the XOR function itself is binary (two arguments), but can be used here as unary (one argument), since one (s) is constant in all applications. That argument will be the eight-bit checksum that characterizes the program.

To set format at 40 columns and literals mode, 9020 and the start of 9030 should be familiar (see 1.3 Interoperability). The rest of 9030 clears the screen for input of initialization data. A string of 39 spaces (b$) is generated in 9050, which also opens the disk command file. A dateline (dt$ 9060) is input using the line-in utility at 8200.

Program planning recognized that we had to translate stored bytes from the program into special characters (within quotes) and commands. Two string arrays are dimensioned for the purpose: q$ within quotes, k$ for commands (9070). The ninety commands available through BASIC 4.0 are contained in DATA statements (10000–10060) and read into k$ at 9070. To translate nonprintable characters, data are read from 10070–10100 into the appropriate parts of q$ at 9080; since the "pi" character prints differently on different printers, chr$(255) is entered separately. Other characters will print as their ASCII values when needed. Note that the DATA statements include values for all machines; only two have different meanings among the machines, and they are entered in their VIC/64 incarnations, not those for CBM.

Now that all predefined data are established, we set a working variable (x 9100) to the amount of FREe memory. We save about 1K for working space, and dynamically allocate the rest to arrays for storing the results of our analysis. The information we want to save is the name of a variable or a line number, and the lines at which it is referenced. For printout, we need up to 5 characters for each field. Therefore, we could squeeze 13 columns of information onto an 80-column page (leaving a blank space between fields), but that would be very crowded. For a clean display, we'll use 11 columns per page—the name of the referenced item and up to ten references per line of printout. We had to solve the print question first in order to design the arrays. The problem is to have enough elements to handle even a very large program in a computer of moderate size. What we will use is one string array (z$) for the names of the referenced items, and an integer array (z%) dimensioned ten wide for the references themselves. Each reference then has up to 6 bytes for its name plus 2 for its address plus twenty for the references to it. To provide a small cushion for garbage collection (expanded VIC and, especially, 64), we model the array as needing 30 bytes; strictly, 28 is enough, but the few extra elements are not likely to be worth the push. Subtracting 31 elements from the arrays assures us of 930 bytes for working storage. (It looks like 32 elements, but remember the zero index.) Finally, we exit to the main entry point of the program, line 100.

It is arguable whether we should name the target program in initialization, carrying the logic in the 9000's through line 170 or even 190. The question is only a matter of taste and judgement, and is of little practical importance. Breaking it here puts most of the one-time operations at high line numbers where they belong. If you wanted to modify the program to permit superlisting more than one program per initialization, the entry is at the right place. If you try that, you will discover that the delay to reinitialize the arrays is so great that it won't be worth the effort.

5.2 SUPERLIST utilities

SUPERLIST is a relatively simple program which needs little user interaction. Many of the utilities here have been carried over from more advanced products because they serve needed functions and have already been checked out; the most efficient SUPERLIST coding would simplify those utilities, at substantial cost in generality. In many cases, SUPERLIST was designed for clarity rather than efficiency. It can be made faster and smaller, but might be less understandable after that kind of "optimization."

Two routines are assigned low line numbers for speed of execution. Lines 20–27 GET a single character from the keyboard. Entered at 20, the buffer is emptied (20) before the character (c$) is awaited (25), then returned with its ASCII value (c). Entry at 25 allows the buffer to hold characters typed in while earlier ones are being digested.

The users of the character-input logic are "Hit a key" (8010), yes/no logic entered at 8050, single-character input (8100—not needed in SUPERLIST), and line-in logic (8200). The logic from 8080 through 8095 is one way to require a RETURN to accept single-character input. It allows the option of deleting the character, but accepts no keys except RETURN and DEL.

Line input (8200–8290) uses character input to permit commas and colons (precluded by INPUT). A line must be ended with a RETURN which is tested at 8220. A deletion from a non-null string is handled in 8230. Along with all other nonprintable characters, DELete of a null string is rejected in 8250. The quote mark is rejected at 8260. The longest input string is 18 characters: a drive number, a colon, and up to 16 characters of program name. (The dateline is arbitrarily limited to the same length.) If the string is less than 18 characters long (8270), an acceptable character is printed and tacked onto the working string (x$). If the string is already its full size, the program continues to wait for a RETURN or DEL (8280).

Two other routines approach the status of utilities. The one that outputs a line and ejects a page (7900–7960) keeps count of lines, then pages as required or on a call. A prefix string (s$) is attached to the print string (p$) to set case and (if you code it that way) to set the left margin. The string is printed and nulled (7920), the line counter (l) is incremented, and the routine returns if the page is not yet full. If all allotted lines have been printed, enough null lines are printed to provide bottom and top margins (7950). That is also the entry for a forced page eject; by testing for l (implicitly, >0), we avoid ejecting a blank page if we have just finished a page of text when the call comes.

The other semi-utility is the routine (30–34) that reads in two bytes, XORs them into the checksum, and creates a number from them for memory address or line number. Note that the logic is complicated by the fact that the ASC function bombs on a null string, while the disk returns a null string for a character zero. This routine really doesn't need the high speed it's given, but it is as logical to put it here as anywhere else—and the speed doesn't hurt. Along the same lines, if you didn't use buffering of input in the line-in logic (8200), you would have to give it smaller line numbers to avoid typing too fast for processing.

5.3 Starting a SUPERLIST

The main entry point (100) ensures that the primary file is closed, then sets up the screen. The program name (pr$) is input through a call to line input (8200 @ 110); clearing the checksum (s) is unnecessary for the program as coded, but it clarifies operations a bit. The program file is opened to read (120). Possible errors trapped at this point include file not found, read error, and even device not present (turn the power on first).

Note that there are many options in entering a program name. You may use the pattern-matching capabilities, but remember that the name is not checked from the directory—what's printed at the top of your SUPERLISTing will use asterisks and question marks if you typed them in. If you wish, you may specify the drive in the name; if you do, the number and colon are deleted (160) between opening the file and printing its name. If you give a blank name (after deleting drive number), disk error won't trap it; the program does so (170) with an appropriate message. Meanwhile (130), an option is offered for graphics printout. The print prefix string (s$) is either null or cursor-down, the character causing literals print, depending on your answer. If you want to put a left margin on your page, add blanks to the definition of s$.

The last operation before starting processing of the individual lines (190) is to complete the display and to read the first two bytes of the program, the start-of-BASIC pointer SAVEd with the program. As written here, the checksum includes that pointer. Consequently, the checksum will vary with the machine that did the SAVE. If you want to be independent of that term, just reset the checksum (s = 0) after GOSUB30 in 190.

Before looking at line-by-line logic, it's worth a moment to check out the thinking behind it. The easy part of the process (given what we've already put into the arrays) is translating all those tokens. The tough part is handling line numbers and variables. We can recognize a variable by the fact that it begins with an alpha character. The only other ways to have alphabetics in program lines are in quotes, remarks, and data statements. Therefore, we need flags for start of a variable, being in quotes mode, being in a remark, and being in a data statement. In contrast, a line number is recognizably numeric, but not all numbers designate lines. A line flag is set after specific commands: GOTO, RUN, GOSUB, and THEN. When a byte other than a digit is found, the line number is complete.

5.4 Processing a line

The first two bytes of a line are the address of its successor—unless it's the line after the end of the program, when they are both nulls. To process a line, we get the first two bytes (1010) and enter wrapup if they can't be the address of the next line. Otherwise (1020), we assign the current line's address to n and the next one's to m. Then we pick up the current line number (ln) and initialize all the flags needed to process the line (qf, df, rf, vf, lf). Note that each flag has a value of 1 if "true" and of "0" if false. Thus, df = 1 in a data statement, rf = 1 in a remark, qf = 1 in quotes mode; vf = 1 if a variable is being named, lf = 1 if a line number *may* be starting next. The last jobs in 1020 are initializing the print string (p$) with the line number (bracketed with blanks) and the reference string (z$).

In order to display on the screen what the computer is doing, 1030 prints the line number. Then the rest of the line is input (x$) in a single FOR . . . NEXT loop. By processing the line as a whole, this version of SUPERLIST has several advantages over the byte-at-a-time version built several years earlier. However, most processing operates on single bytes, so the rest of line processing (1000–1800) is essentially within a FOR . . . NEXT loop indexed by i through the length of x$. We process each byte in terms of its ASCII value (c), first by XORing it into the checksum (s), then by reconstructing its character (c$). While later processing may alter c$, a REMark leaves it unchanged for printing. If the current character is a quote (1050), the quotes flag is toggled and no further processing occurs. The quote is one of many cases where syntax errors could be caught but aren't. A line:

```
20 a=b"
```

will be a syntax error since the variable (b") is illegal. But it can be stored, and SUPERLIST will not flag it. If you wish, you can expand SUPERLIST to check for all sorts of syntax errors instead of having to run test cases—in that way, you could accomplish many of the verification functions of a compiler.

Information in quotes is processed separately (1060), as are commands (1080). The code in the 1100's processes unshifted characters not in quotes or remarks. If the line flag (lf) is set and the current character is numeric, the character is tacked onto the string (z$ 1100). If the character is not numeric, line-reference is ended by calling 1600 (1110); other than for a syntax error, this handles the cases of comma (ON . . . GOTO) and a variable after THEN. If a variable is currently flagged (1130) and the current character is alphameric, it is tacked on. Although a variable usually ends with a command or end of line, it can be terminated by a comma or right parenthesis; they are handled by calling 1500 (1140). The variable flag (vf) is set (1150) if the character is alpha and no flag is set. Finally (1160), a colon resets the data flag (df), since that is one way to end a DATA statement. (Note that only the end of a line terminates a REMark.)

Within quotes (1400–1420), the string from the quote array (q$) is assigned to c$ for printing. Ending a variable (1500–1520) begins by chopping off extra characters (1510) if there were more than two, then tacks on the current character if it is part of the variable ($, %, or left parenthesis). If it was $ or %, then a left parenthesis is also added if it is next on the line (1520). The rest of the processing matches that for variables (1600–1690).

Since the line flag (lf) was set when the leading command was given, intervening spaces may occur on a line number. They are handled in 1600—essentially by ignoring them. A null line number is possible after THEN; in that case (1610), the flag is reset before exiting. This would also be a good point at which to check for legality of line number (val(z$) < 64000) if you wish. Now a counter (q) is used instead of a flag to identify one of three conditions: a new line of ten entries is found for the reference (q = 0); an old line has room for it (q = 9); or there is no place left to put it (q = 5). The counter is initialized (1620) and a loop initiated (index j) among the references already encountered (z$ array). If we get through the used entries without success, we use a new one (1620). If we find the name in the list (1630) with space for the entry, we use that. If there is no room (1640), we report the problem and continue with the rest of the work. In 1650, the program finds the first available place to put the entry. The reference that is stored (z% 1660) is complex. An integer array can count from about −32K to +32K. Line numbers range from 0 through 63999, so they need to be massaged before insertion. The easy way would be simply to subtract a suitable number, like 32767, from the line number and store the result. The problem is that that would let you store a 0 (for a true line 32767). That's the value when the array is initialized, so it could not be detected easily. Therefore, we bias the value by adding 1 (subtracting −1) if the stored value is not negative.

The last major operation to end reference processing is to reset the line-number flag (1680) if the current character is not a comma (remember ON . . . GOTO and ON . . . GOSUB). Finally, the variable flag is always reset and the reference string cleared before the return (1690).

Processing a command is surprisingly simple. If the variable flag was set, the variable is wrapped up; if the line flag was set, the line number is wrapped up. Then the current character is replaced by the string in the command array (k$). Note that using ON . . . GOSUB allows all this and more to be done in one line (1710). If the command is GOTO, RUN, GOSUB, or THEN, the line flag (lf) is set. If the command is DATA, the data flag (df) is set. Finally, if the command is a remark, we set the remark flag (rf 1740).

At this point (1800), we have completed processing that byte, and c$ has the information to be added to the print string. If there's no room to attach c$ and keep an 80-character print line, the existing line is printed (and nulled) by calling 7900. The contribution from the current byte is attached (1810) before the next byte is processed; when the line is finished, variables and lines are wrapped up if needed, the remaining print line is output, and the next line is called for by moving back to 1000.

5.5 Printing the references

When the next line is at an impossible address, the processing of lines exits to wrapup (2000). The disk file is closed (2005), the display is prepared for references, and a fresh flag (f) is cleared to indicate that line references are to be printed. A blank line is printed (2010), then the checksum; a routine that puts out a new page with a header (2900) is called to start the variable references. Rather than searching all of the possible entries (v), we first find the last entry (q 2020). The work string (x$) is set larger than any reference name (2040); anything after "z" would do, but chr$(255) is impressive. The list of string names is scanned (2040–2050) to find the smallest one still in it; when all have been output, the program goes to 2500 for the final operations—finishing the page, ejecting a page at user option, and saying "Goodbye!"

Before putting out the line, a check is made (2060) for the first variable to be printed. To maximize interoperability, lines are formatted instead of using printer capabilities. Columns are right-justified for easy reading, beginning with the reference's name or number (2070). The reference is effectively removed from the array (z$) by setting it to chr$(255). For each non-null entry in the reference array (z%), the print string (p$) is extended (2080). When the string is complete (2090), it is printed and the next reference is sought.

You may want to modify the program in the area of printing line-number references. One look at the output shows you that the numbers are in ASCII order, not numerical. That means that references to line 30 are displayed after those to 1000. By using the VAL function, you can order them numerically. However, you must remember that there can be a line 0, and you will find the logic several lines longer than that given here. Similarly, you might investigate using separate pages for the different types of variables and for arrays. All the information you need to make the changes is here, but you may find doing it more complex than you expect.

5.6 What doesn't work

SUPERLIST has a number of anomalies which could be distracting in some applications. Correcting them is straightforward—usually requiring just setting and testing flags in appropriate places. But testing those flags will take time, and whether it's worthwhile depends on the application. The obvious limitations of the program as documented are listed below.

The command TO is unique in Commodore BASIC. It designates the upper bound of a FOR definition, and it completes a two-word command: GO TO. SUPERLIST doesn't have a flag set when GO is encountered and tested when TO is found. GO TO is functionally identical with GOTO, and is followed by a line number; otherwise, TO is followed by a number or a variable. Since GO TO offers no advantage over GOTO and takes two extra bytes, the omission is probably not significant.

If you set a flag for a number, then you can catch the fact that "e5" is not a variable in "1e5." As the program is written, you will see a spurious variable.

A user-defined function introduces a spurious array "variable." SUPERLIST itself uses FNX (defined in 9010) for the XOR operator. Again, a flag could be set after the FN command to indicate that the next character is not part of a true variable; on the other hand, if you don't use a real X(array, having the program show you where FNX is used may be constructive.

BASIC 4.0 has a set of unique expressions which give rise to spurious "variables" in SUPERLIST. (In fact, it was failure to consider all of them that led to the bug in formatting a disk with a variable id.) When a file is OPENed under 2.0, all the special information needed is put in quotes, as though it were part of the name. DOPEN the same file (BASIC 4.0), and you put that information into the line itself. As a result, one can find spurious "variables" such as "d1," "w," and "1123" in a SUPERLIST. In a HEADER command, you can generate a spurious "ix4" if you format the disk within the program. The solution is straightforward: a BASIC 4.0 flag, set on any appropriate command, suppressing variables except within parentheses, and cleared at the next command.

LIST presents an anomaly in its use of line numbers. LIST 123-456 is not SUPERLISTed as having line references at all. There are two reasons for that: neither 123 nor 456 is really a line number, and LIST does not belong in finished code. The numbers following LIST and the dash are bounds for line numbers; if the numbers don't exist, no error is generated. Consequently, omitting the logic for the LIST command is a good design decision, not simply a convenience.

Commodore offers a set of special characters which are not SUPERLISTed. They are the graphics characters which do not convert to uppercase letters; on a PET, they include shifted numerals and punctuation. On CBM, they are not accessible from the keyboard at all. On a 64, they are reached with the Commodore key. Since there is no obvious notation for those characters, the version of SUPERLIST shown here will print question marks for them. If you have a preferred notation, just enter it in the appropriate DATA statements.

Since SUPERLIST is in BASIC, you have the option to supply the missing logic for any or all of the above. If your code is not interoperable, you will probably want to use BASIC 4.0. Then add the flag and the logic to do the job you want. Trap GO TO if it's worth the effort; figure a solution for LIST if you think that that would be worthwhile. But recognize that those changes will slow the program further and will enlarge it too.

When you have finished with your own version of SUPERLIST, you may want to compile it for your primary computer. In BASIC, it runs slowly. Most of the time is spent finding places to put variables and line numbers. A large program may take 30 minutes or more to SUPERLIST in BASIC but be limited by the printer after compilation—to 5 minutes or so. If you plan to compile, then the extra time required to handle the special cases above won't be significant.

5.7 SUPERLISTing

SUPERLIST was modified before it was run to make the listing in this section. The changes indicate what you may want to do on rare occasion, or an option you might want to build in for yourself. The difference between this listing and an "ordinary" one is that this was intended for publication.

Getting wide margins is simple enough. Left margin is set by adding spaces to s$ in 130. The right margin is set indirectly—by changing the number of characters per line (set at 80 in 1800). To get 6-character margins, add six blanks to s$ and set the line limit at 68 in 1800. Of course, references can still take 66 characters. The 8300 printer has a number of features that will surprise you if you're familiar with the dot-matrix variety. The one of immediate interest is that its underscore character is not the shifted "$" but the left arrow. The shifted "$" prints as "$." We can't just put the correction into the q$ array since it would appear in brackets. This listing was generated by adding the line:

```
1415 IFC=164THENC$="
```

followed by a left arrow. Note that the arrow simply isn't on the print wheel; you can't get there from here. Since it can only appear legally in quotes, you could add it to the q$ array. The same thing could be done for other special characters, and you could build a version of SUPERLIST just for formal printing on the 8300.

Some problems with printers have no practical solutions. The easiest demonstration is the character "pi," which can appear on a program line. It looks like a command and in some senses behaves like one. If you put a special trap in the command logic (1700's), you will slow down the program significantly. For most purposes, the slowdown isn't worth the small problem of interpolating a pi. If you disagree, write your own version.

There are some obvious variations on the theme of SUPERLIST that are included on the disk you will get on returning the card. CHECKSUM is a fast and simple program that computes an 8-digit checksum matching the one from SUPERLIST. It will run on a VIC if you remove the formatting. PROGRAM COMPARE reports the line numbers that differ between two programs; it is useful in identifying different versions when they have different checksums. SUPERLIST takes about 20 minutes to SUPERLIST itself. It is CHECKSUMmed in about 2, and can be compared with another version in about 6. If you want to compile CHECKSUM or SUPERLIST, note that some compilers won't handle the dynamic dimensioning used in those programs. A compiled SUPERLIST can be limited in speed by the printer—it should run about five times as fast as the interpreted version.

```
superlist                                   Operational

 10 goto9000:rem (c) 1982 by m richter 90064
 20 getc$:ifc$goto20:rem empty buffer
 25 getc$:ifc$thenprint" <c-l>";:c=asc(c$):return
 27 print"_<c-l>";:goto25
 30 get#1,c$:x=0:ifc$>""thenx=asc(c$):s=fnx(x)
 32 get#1,c$:c=0:ifc$>""thenc=asc(c$):x=x+256*c:s=fnx(c)
 34 return
 100 close1:print"<clr>","<rvs> SUPERLIST ":rem main entry
 110 s=0:print"<c-d>Program name?":gosub8200:pr$=x$
 120 open1,8,0,pr$:input#8,x,x$:ifxthenprintx$:gosub8000:goto100
 130 print"<c-d>Graphics printout";:gosub8050:s$="":ifithens$="
<c-d>
 160 ifmid$(pr$,2,1)=":"thenpr$=mid$(pr$,3)
 170 ifpr$=""thenprint"<c-d>The program needs a name":gosub8000:
goto100
 180 open4,4:print#4,s$pr$left$(b$,40-len(pr$))"<c-d>"dt$:print#4:
l=2
 190 print"<c-d><rvs>current line<c-d>":gosub30:m=x:rem start of b
asic
 1000 rem process a line
 1010 gosub30:ifx<1024goto2000:rem end
 1020 n=m:m=x:gosub30:ln=x:qf=0:df=0:rf=0:vf=0:lf=0:p$=str$(ln)+"
":z$="
 1030 print"<c-u>"ln:x$="":fori=n+4tom-2:get#1,c$:x$=x$+c$:next:
get#1,c$:rem line in
 1040 fori=1tolen(x$):c=asc(mid$(x$,i)):s=fnx(c):c$=chr$(c):ifrf
goto1800:rem rem
 1050 ifc=34thenqf=1-qf:goto1800:rem quote mark
 1060 ifqfgoto1400:rem in quotes
 1080 ifc>127goto1700:rem command
 1100 iflfthenifc>47andc<58thenz$=z$+c$:goto1800:rem add digit
 1110 iflfthengosub1600:rem end line
 1130 ifvfthenif(c>47andc<58)or(c>64andc<91)thenz$=z$+c$:goto1800:
rem add char.
 1140 ifvfthengosub1500:rem end variable
 1150 ifc>64andc<91thenvf=1-df:z$=c$
 1160 ifc=58thendf=0:rem colon
 1170 goto1800
 1400 rem within quotes
 1410 ifq$(c)>""thenc$="<"+q$(c)+">
 1420 goto1800
 1500 rem end of variable
 1510 z$=left$(z$,2):ifc=36orc=37orc=40thenz$=z$+c$:rem $ % (
 1520 if(c=36orc=37)andmid$(x$,i+1,1)="("thenz$=z$+"(
 1600 ifc=32andz$=""thenreturn:rem leading space in line reference
 1610 ifz$=""goto1680
 1620 q=5:forj=0tov:ifz$(j)=""thenz$(j)=z$:q=0:k=j:j=v:rem new slo
t
 1630 ifz$(j)=z$andz%(j,9)=0thenq=9:k=j:j=v:rem old slot
 1640 nextj:ifq=5thenprint"<rvs> File full <c-u>":goto1680
 1650 forj=0to9:ifz%(k,j)=0thenq=j:j=9
 1660 nextj:z%(k,q)=ln-bs-(ln>=bs)
 1680 ifc<>44thenlf=0:rem not comma
 1690 vf=0:z$="":return
 1700 rem command
 1710 onvfgosub1500:onlfgosub1600:c$=k$(c-128)
 1720 ifc=137orc=138orc=141orc=167thenlf=1:z$="
 1730 ifc=131thendf=1:rem data
 1740 ifc=143thenrf=1:rem remark
 1800 iflen(p$)+len(c$)>80thengosub7900
 1810 p$=p$+c$:nexti:onvfgosub1500:onlfgosub1600:gosub7900:goto100
0
 2000 rem wrapup
 2005 close1:print"<c-u><c-u>"b$:printb$:print"<c-u><rvs>Now print
ing<c-d>":f=0
 2010 gosub7900:p$="<c-d> CHECKSUM:"+str$(s):gosub7900:p$="<c-d>Li
ne-number":gosub2900
 2020 fori=vto0step-1:ifz$(i)>""thenq=i:i=0
 2030 nexti
 2040 x$="<pi>":fori=0toq:ifz$(i)<x$thenj=i:x$=z$(i)
 2050 nexti:ifx$="<pi>"goto2500
 2060 iff=0andasc(x$)>57thenp$="<c-d>Variable":gosub2900:f=1
 2070 p$=left$(b$,6-len(x$))+x$:z$(j)="<pi>":print"<c-u>"p$:fori=0
to9
 2080 x=z%(j,i):ifxthenx$=str$(x+bs+(x>0)):p$=p$+left$(b$,6-len(x$
))+x$
 2090 nexti:gosub7900:goto2040
 2500 rem the end
 2510 gosub7950:print"<c-d>Eject a page";:gosub8050:ifi=0thenl=.1:
gosub7950
 2520 close4:print"<c-d>Goodbye!":end
 2900 gosub7950:p$=p$+" references":gosub7900:goto7900:rem page wi
th header
 7900 rem line out
 7910 ifp$>""thenp$=s$+p$
 7920 print#4,p$:p$="":l=l+1:ifl<l0thenreturn
 7950 iflthenforl=ltolp-1:print#4:next:l=0:rem page with blanks
 7960 return
 8000 rem utilities
 8010 print"<c-d><rvs> Hit a key to continue";:gosub20:print:print
"<clr>":return
 8050 x$="yn
 8060 print"? ";
 8070 gosub20:fori=1tolen(x$):ifc$<>mid$(x$,i,1)thennext:goto8070
 8075 i=i-1:printc$;
 8080 getw$:ifw$=""thenprint"_<c-l>";:goto8080
 8085 print" <c-l>";:ifw$=r$thenprint:return
 8090 ifw$=chr$(20)thenprintw$;:goto8070
 8095 goto8080
 8200 x$="":rem input a string
 8210 getc$:ifc$goto8210
 8220 gosub25:ifc=13thenprint:return
 8230 ifc=20andx$>""thenprintc$;:x$=left$(x$,len(x$)-1):rem delete
 8250 if(127andc)<32goto8220:rem cursor control characters
 8260 ifc=34goto8220:rem quote
 8270 iflen(x$)<18thenprintc$;:x$=x$+c$
 8280 goto8220
 9000 rem initialize
 9010 lp=66:l0=60:r$=chr$(13):bs=32767:deffnx(i)=(sori)andnot(sand
i)
 9020 print"<clr><grph><home><home><tset><c-d><c-d><c-d><c-d><c-d>
<c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d>
<c-d><c-d><c-d><c-d><c-d><c-d><c-d>"spc(39)"<bset><wht>":poke59468
,14:rem format
 9030 poke53272,2orpeek(53272):print"<clr>","<rvs> SUPERLIST "
 9050 b$="                  ":b$=b$+b$+b$:open8,8,15
 9060 print"<c-d>Dateline?":gosub8200:dt$=x$
 9070 dimk$(127),q$(255):fori=0to90:readk$(i):next
 9080 fori=0to31:readq$(i):next:fori=128to160:readq$(i):next:q$(25
5)="pi
 9100 x=fre(0):ifx<0thenx=x+65536
 9110 v=int(x/30-32):dimz$(v),z%(v,9)
 9190 goto100
 10000 dataend,for,next,data,input#,input,dim,read,let,goto,run,if
,restore
 10010 datagosub,return,rem,stop,on,wait,load,save,verify,def,poke
,print#,print
 10020 datacont,list,clr,cmd,sys,open,close,get,new,tab(,to,fn,spc
(,then,not
 10030 datastep,+,-,*,/,"<pi>",and,or,>,=,<,sgn,int,abs,usr,fre,po
s,sqr,rnd,log
 10040 dataexp,cos,sin,tan,atn,peek,len,str$,val,asc,chr$,left$,ri
ght$,mid$,go
 10050 dataconcat,dopen,dclose,record,header,collect,backup,copy,a
ppend
 10060 datadsave,dload,catalog,rename,scratch,directory
 10070 data?,?,stop,?,?,wht,bell,?,dis,enab,l-f,?,?,c/r,lit,tset,?
,c-d,rvs,home
 10080 datadel,ldel,-end,?,?,sc-u,?,esc,red,c-r,grn,blu
 10090 data?,?,?,run,?,f1,f3,f5,f7,f2,f4,f6,f8,^c/r,grph,bset,blk,
c-u,off,clr
 10100 datainst,lins,-st,?,?,sc-d,?,?,pur,c-l,yel,cyn,^spc
```

CHECKSUM: 187

```
Line-number references

  100   120   170  9190
 1000  1810
 1400  1060
 1500  1140  1710  1810
 1600  1110  1710  1810
 1680  1610  1640
 1700  1080
 1800  1040  1050  1100  1130  1170  1420
   20    20  8010  8070
 2000  1010
 2040  2090
   25    27  8220
 2500  2050
 2900  2010  2060
   30   190  1010  1020
 7900  1800  1810  2010  2010  2090  2900  2900
 7950  2510  2510  2900
 8000   120   170
 8050   130  2510
 8070  8070  8090
 8080  8080  8095
 8200   110  9060
 8210  8210
 8220  8250  8260  8280
 9000    10

Variable references

   b$   180  2005  2005  2070  2080  9050  9050  9050  9050  9050
   bs  1660  1660  2080  9010
    c    25    32    32    32    32  1040  1040  1040  1050  1080
    c  1100  1100  1130  1130  1130  1130  1150  1150  1160  1410
    c  1410  1510  1510  1510  1520  1520  1600  1680  1710  1720
   c  1720  1720  1720  1730  1740  8220  8230  8250  8260
  c$    20    20    25    25    25    30    30    30    32    32
  c$    32  1030  1030  1030  1040  1100  1130  1150  1410  1510
  c$  1710  1800  1810  8070  8075  8210  8210  8230  8270  8270
  df  1020  1150  1160  1730
 dt$   180  9060
   f  2005  2060  2060
   i   130  1030  1040  1040  1520  1810  2020  2020  2020  2020
   i  2030  2040  2040  2040  2040  2050  2070  2080  2090  2510
   i  8070  8070  8075  8075  9010  9010  9010  9070  9070  9080
   i  9080  9080  9080
   j  1620  1620  1620  1620  1620  1630  1630  1630  1630  1640
   j  1650  1650  1650  1650  1660  2040  2070  2080
   k  1620  1630  1650  1660
 k$(  1710  9070  9070
   l   180  2510  7920  7920  7920  7950  7950  7950  7950
  lo  7920  9010
  lf  1020  1100  1110  1680  1710  1720  1810
  ln  1020  1020  1030  1660  1660
  lp  7950  9010
   m   190  1020  1020  1030
   n  1020  1030
  p$  1020  1800  1810  1810  2010  2010  2060  2070  2070  2080
  p$  2080  2900  2900  7910  7910  7910  7920  7920
 pr$   110   120   160   160   160   170   180   180
   q  1620  1620  1630  1640  1650  1660  2020  2040
 q$(  1410  1410  9070  9080  9080  9080
  qf  1020  1050  1050  1060
  r$  8085  9010
  rf  1020  1040  1740
   s    30    32   110  1040  2010  9010  9010
  s$   130   130   180  7910
   v  1620  1620  1630  2020  9110  9110  9110
  vf  1020  1130  1140  1150  1690  1710  1810
  w$  8080  8080  8085  8090  8090
   x    30    30    30    32    32   120   120   190  1010  1020
   x  1020  2080  2080  2080  2080  9100  9100  9100  9100  9110
  x$   110   120   120  1030  1030  1030  1040  1040  1520  2040
  x$  2040  2040  2050  2060  2070  2070  2080  2080  2080  8050
  x$  8070  8070  8200  8230  8230  8230  8230  8270  8270  8270
  x$  9060
  x(    30    32  1040  9010
  z$  1020  1100  1100  1130  1130  1150  1510  1510  1510  1510
  z$  1520  1520  1600  1610  1620  1630  1690  1720
 z$(  1620  1620  1630  2020  2040  2040  2070  9110
 z%(  1630  1650  1660  2080  9110
```

5.8 SYS CHECKSUM

A fast checksum program interoperable on all machines is clearly desirable. The one is BASIC 2.0 on the demonstration disk which was developed from SUPERLIST by stripping out all unnecessary code and putting the XOR function in line. Then an alternative version, SYS CHECKSUM, was written on the framework of CHECKSUM. It differs in using a machine-language routine to perform the XOR operation. Three factors prompted the change. The desire to:

- Demonstrate using machine language within BASIC.
- Speed up the program (about 20%).
- Provide a display of the program's activity.

The logic for providing the XOR function is trivial in machine code. The 6502 family, like most microprocessors, offers the needed instruction (EOR) in its set of primitives, with several addressing modes. The program POKEs the value to be XOR'd into the byte following the EOR-immediate instruction ($49). The rest of the "routine" is simply loading the checksum into the accumulator before executing the EOR, then storing the accumulator into the checksum's location. By putting the checksum into a blank cell of the screen, we have a visible display of the program's progress. We could put that display wherever we wished. Placing it one pixel right of the top left corner of the screen is convenient. Putting it into the corner would be a problem: BASIC line 10 would have two bytes = 0 (the low-order of the checksum address). If you LISTed the program after RUNning it, you would find strange line numbers, which would be confusing. BASIC would still run correctly, since it uses the next-instruction address instead of looking for the end-of-line marker. Still, the 0's would be inconvenient.

For the 64, the machine code is:

```
LDA $0401
EOR #($1016)
STA $0401
RTS
```

For PET or CBM, only the addresses are changed:

```
LDA $1001
EOR #($0416)
STA $1001
RTS
```

We can identify the type of computer by noting that start of BASIC on the PET or CBM coincides with the start of the normal screen position on the 64. Our programs control both what's in BASIC and what's on the screen.

In initialization, we have cleared the screen (9020) before we look at 1035 (9040). On the 64, we'll find 160, a reversed space. On PET or CBM, we'll find 143, the first REM in line 10. Next (9050), we read into the nine bytes following that REM the machine-language XOR program, then open the command channel to the disk and set s to the checksum site for 64, and offset l to the location of the routine. If l is <2000 (actually, 1040), we reset the checksum site and POKE for literals on PET/CBM (9060). Note that initialization may POKE any value after the EOR; the program puts in the ASCII of each character as we get it. However, if we initialize to 0, the line would not list correctly until reading began. (We only POKE in values for non-null characters, so we will never have a 0 after processing starts.)

Since it only takes two lines to process the whole file, they're put right up at the top of the program (12–14). Of course, we don't have to XOR a 0 into the checksum. On the other hand, we want to look for three consecutive 0's to find the end of the program. Line 12 handles both jobs quickly. In 14, the ASCII value of the current character is POKEd into its location (l), then the routine is called with SYS(s). It really is that easy.

We could have used the USR function, but that requires moving the argument from the floating-point accumulator in memory to the processor's accumulator register. That means extra code and another location to select depending on the computer in use. And it offers no advantage over SYS in this case. There are more alternative designs in a typical machine-language routine than in its BASIC equivalent, and selecting among them is seldom as simple as choosing SYS over USR in this program. The 20% speedup that machine language gives in CHECKSUM would probably not be worth the time to write the extra code if the program's purpose were simply to compute a checksum. The effort is justified by its other function—as an example of machine code embedded in a BASIC program.

```
sys checksum                              Operational

 10 goto9000:rem           :rem (c) 1982 by m richter 90064
 12 get#1,c$:ifc$=""thenget#1,c$:ifc$=""thenget#1,c$:ifc$=""then
close1:return
 14 pokel,asc(c$):sys(x):goto12
 20 getc$:ifc$goto20:rem empty buffer
 25 getc$:ifc$thenprint" <c-l>";:c=asc(c$):return
 27 print"_<c-l>";:goto25
 100 close1:print"<clr>","<rvs> CHECKSUM ":rem entry
 110 print"<c-d>Program name?":gosub8200:pr$=x$
 120 open1,8,0,pr$:input#8,x,x$:ifxthenprintx$:gosub8000:goto100
 160 ifmid$(pr$,2,1)=":"thenpr$=mid$(pr$,3)
 170 ifpr$=""thenprint"<c-d>The program needs a name":gosub8000:
goto100
 1000 rem the works
 1010 x=l-4:pokes,0:gosub12:print"<c-d>Checksum="peek(s)
 1030 print"<c-d>Another program";:gosub8050:ifithenprint"<c-d>Goo
cbye!":end
 1040 goto100
 8000 rem utilities
 8010 print"<c-d><rvs> Hit a key to continue";:gosub20:print:print
"<clr>":return
```

```
 8050 x$="yn
 8060 print"? ";
 8070 gosub20:fori=1tolen(x$):ifc$<>mid$(x$,i,1)thennext:goto8070
 8075 i=i-1:printc$;
 8080 getw$:ifw$=""thenprint" <c-l>";:goto8080
 8085 print" <c-l>";:ifw$=chr$(13)thenprint:return
 8090 ifw$=chr$(20)thenprintw$;:goto8070
 8095 goto8080
 8200 x$="":rem input a string
 8210 getc$:ifc$goto8210
 8220 gosub25:ifc=13thenprint:return
 8230 ifc=20andx$>""thenprintc$;:x$=left$(x$,len(x$)-1):rem delete
 8250 if(127andc)<32goto8220:rem cursor control characters
 8260 ifc=34goto8220:rem quote
 8270 iflen(x$)<18thenprintc$;:x$=x$+c$
 8280 goto8220
 9000 rem initialize
 9020 print"<clr><grph><home><home><tset><c-d><c-d><c-d><c-d><c-d>
<c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d>
<c-d><c-d><c-d><c-d><c-d><c-d><c-d>"spc(39)"<bset><wht>":poke53272
,2orpeek(53272)
 9040 l=1024:ifpeek(l+11)<>143thenl=4096:rem 1035 in basic/screen
on pet/64
 9050 fori=l+12tol+20:readc:pokei,c:next:open8,8,15:s=1025:l=l+16
 9060 ifl<2000thenpokel-2,128:pokel+3,128:poke59468,14:s=32769:rem
 for 64
 9090 goto100
 10000 data173,1,4,73,1,141,1,4,96:rem lda & eor # sta & rts (&)
CHECKSUM: 167
```

```
Line-number references

   100   120   170  1040  9090
    12    14  1010
    20    20  8010  8070
    25    27  8220
  8000   120   170
  8050  1030
  8070  8070  8090
  8080  8080  8095
  8200   110
  8210  8210
  8220  8250  8260  8280
  9000    10
```

```
Variable references

     c    25  8220  8230  8250  8260  9050  9050
    c$    12    12    12    12    12    12    14    20    20    25
    c$    25    25  8070  8075  8210  8210  8230  8270  8270
     i  1030  8070  8070  8075  8075  9050  9050
     l    14  1010  9040  9040  9040  9050  9050  9050  9050  9060
     l  9060  9060
   pr$   110   120   160   160   160   170
     s  1010  1010  9050  9060
    w$  8080  8080  8085  8090  8090
     x    14   120   120  1010
    x$   110   120   120  8050  8070  8070  8200  8230  8230  8230
    x$  8230  8270  8270  8270
```

5.9 TURTLEWALK

Advanced software in BASIC is used for many purposes: business, games, education—any programming application. Where SUPERLIST is an example for software development and the following chapter deals with a business application, a few pages are worthwhile for an educational program that borders on a game.

The concept of a program is difficult to teach. The idea is an abstraction, like a "function" in algebra, and is not self-evident as apples, lines of BASIC, or other tangibles are. Most modern languages are ill suited to teaching the concept, however easy they are to use when the idea is understood. PILOT is one language in which the idea comes through easily, but it has few applications. Let's construct a "language" in BASIC that teaches the idea in a PILOT-like way. We'll introduce the concept and make it useful and entertaining. We don't want a language that resembles BASIC or FORTRAN, since we want to talk about only the ideas of programming, not a particular implementation. We need a few simple "instructions" that will be understandable to anyone. We'll use a minimum of jargon to build a language and to give the user some exercise with it.

TURTLEWALK uses a figure for a "turtle" that has a recognizable direction. It has "instructions" to point it in the cardinal directions, to move it forward, and to locate it on a screen which may be cleared. Of the many types of programming language, we'll use macros. Like a user-defined function, a macro is an expression in the language which can be used as a command.

Implementing TURTLEWALK is simple enough. The instructions are extensive, and just about fill the 9000's after brief initialization. Since we are trying only to teach the concepts, we don't need any real computing power. As a result, we can limit the system to ten "programs" and can expand each one to its primitives. If we wanted to build a language for use, we would allow more macros and would save memory by deferring their expansion to execution time. That program would run slower, but would still be acceptable to the user. The program might interpret the macro as a number when the assignment was recognized, then execute the numbered routine when it was run. Since BASIC does not offer direct access to the stack, nesting would have to be limited. More to the point, special code would be needed to prevent infinite loops. The language might be "better" that way, but not in terms of its purpose.

The TURTLEWALK language has elements of many standard ones, including BASIC, PILOT, FORTRAN, and FORTH. It is artificial and has no other application. The only reason for its existence is TURTLEWALK, so there is no point to extending or expanding it. Consequently, the language and the program are built for the one purpose, and should be evaluated only in terms of how well they accomplish it. TURTLEWALK might be the first program of a series teaching BASIC, FORTRAN, or any other language, or even teaching the role of the computer in society.

```
turtlewalk                                        Operational

 10 goto9000:rem m richter
 20 getc$:ifc$goto20
 22 getc$:ifc$=""thenprint" <c-l>";:goto22
 24 c=asc(c$):print" <c-l>";:return
 30 x$="neswfchq":ifp$=""thenreturn:rem execute p$
 32 fork=1tolen(p$):c$=mid$(p$,k,1):gosub8110:ifj>6thenk=256
 34 ifj=5orj=6thenx=1:y=1:ifj=5thenprint"<clr><c-d> ";:rem clear h
ome
 36 ifj=4thengosub8200
 38 ifj<4thend=j
 39 printe$t$(d);:nextk:return
 40 return
 50 getc$:ifc$goto50
 52 getc$:ifc$=""goto52:rem no cursor
 54 return
 100 print"<clr><c-d>      <rvs> turtle "
 110 print"<c-d>type in your command or your program
 120 print"<c-d>ending with <c/r>.<c-d>
 130 a=-1:p$="":gosub8300
 140 ifasc(x$)=39goto1000:rem ? assign ?
 150 ifx$=""goto2000:rem wrapup
 160 c=asc(x$):ifc=39goto300:rem macro
 165 ifc=32thenprint" ";:x$=mid$(x$,2):goto150:rem space is ok
 170 ifk%(c)=0goto500:rem invalid
 180 iflen(p$)<255thenc$=chr$(c):printc$;:p$=p$+c$:x$=mid$(x$,2):
goto150
 190 print"<rvs>^":print"<c-d>too much to remember!":gosub8000:
goto100
 300 rem program name
 310 j=0:fori=2tolen(x$):ifasc(mid$(x$,i))=39thenj=i:i=256
 320 next:ifj<3goto500
 330 y$=mid$(x$,2,j-2):x$=mid$(x$,j+1)
 340 j=-1:fori=0to9:ifp$(0,i)=y$thenj=i:i=10
 350 next:ifj<0goto500
 360 iflen(p$)+len(p$(1,j))>255goto190
 370 print"'"y$"'";:p$=p$+p$(1,j):goto150
 500 rem invalid character
 510 print"<rvs>^":print"<c-d>try again":gosub8000:goto100
 1000 rem test assignment
 1010 j=0:fori=2tolen(x$):ifasc(mid$(x$,i))=39thenj=i:i=256
 1020 next:ifj=0goto500:rem no next '
 1030 ifmid$(x$,j+1,1)<>"="goto330:rem in-line
 1100 rem effect assignment
 1110 y$=mid$(x$,2,j-2):ify$=""thenprint"<c-d>it needs a name":
goto1190
 1120 x$=mid$(x$,j+2):ifx$=""thenprint"<c-d>i can't remember nothi
ng!":goto1190
 1130 j=-1:fori=0to9:ifp$(0,i)=""orp$(0,i)=y$thenp$(0,i)=y$:j=i:i=
9
 1140 next:ifj<0thenprint"<c-d>i can't remember any more!":goto119
0
 1150 a=j:print"'"y$"'=";:goto150
 1190 gosub8000:goto100
 2000 rem wrapup
 2010 ifa=-1thenp$="h"+p$:gosub8040:goto100:rem direct
 2020 p$(1,a)=p$:goto100
 2500 rem effect assignment
 8000 rem utilities
 8010 print"<c-d><rvs> please touch a key to continue";:goto20
 8040 gosub30:gosub9990:print:print"<c-d><c-d>do it again";:gosub8
050:ifj=0goto8040
 8045 return
 8050 print"? ";:x$="yn
 8100 gosub20
 8110 fori=1tolen(x$):ifc$=mid$(x$,i,1)thenj=i-1:i=256
 8120 next:ifi<257goto8100
 8130 return
 8200 printe$:ifd=0theny=y+(y>1)
 8210 ifd=1thenx=x-(x<38)
 8220 ifd=2theny=y-(y<23)
 8230 ifd=3thenx=x+(x>1)
 8240 printleft$(d$,y+1)spc(x);:return
 8300 x$="":rem input a line
 8310 getc$:ifc$goto8310:rem clear buffer
 8320 gosub22:ifc=13thenprint:return
 8330 ifc=20andx$>""thenx$=left$(x$,len(x$)-1):printc$;:goto8320
 8340 ifc<>34thenif(cand127)>31thenprintc$;:x$=x$+c$
 8350 goto8320
 8400 gosub50:fori=1tolen(x$):ifc$=mid$(x$,i,1)thenj=i-1:i=256
 8410 next:ifi<257goto8400
 8420 return
 9000 rem initialization
 9010 print"<clr><grph><home><home><tset><c-d><c-d><c-d><c-d><c-d>
<c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d>
<c-d><c-d><c-d><c-d><c-d><c-d><c-d>"spc(39)"<bset><wht>":poke59468
,12:rem format
 9020 dimt$(3):fori=0to3:readt$(i):next
 9030 e$="<c-u> <c-d> <c-l><c-l><c-l>  <c-d><c-l> <c-u><c-l>":d$="
<home><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d>
<c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d>
 9040 dimp$(1,9):fori=0to2:readp$(0,i),p$(1,i):next
 9050 dimk%(255):x$="cefhnqsw":fori=1tolen(x$):k%(asc(mid$(x$,i)))
=1:next
 9080 print"<clr><c-d>     <rvs> turtle ":print"<c-d>do you want i
nstructions";
 9090 gosub8050:ifjgoto100
 9100 print"<clr><c-d>hi!  i'm a turtle   "t$(0)
 9110 print"<c-d>i can face in four directions:
 9120 gosub9990:print"<c-d><c-d>   north  "t$(0)
 9130 gosub9990:print"<c-u>   south  "e$t$(2)
 9140 gosub9990:print"<c-u>   east   "e$t$(1)
 9150 gosub9990:print"<c-u>or west   "e$t$(3)
 9160 gosub9990:print"<c-d><c-d>to tell me to go north, type n.
 9170 print"<c-d>for south, s, for east, e, for west, w.<c-d>
 9180 gosub8000
 9200 print"<clr><c-d>let's try it now.  just type n, s, e, w
 9210 print"<c-d>or q for quit when you want to go on.
 9220 print"<c-d><c-d>           ";:j=0:x$="neswq
 9230 print"<c-l><c-l><c-l>"mid$(x$,j+1,1)"  "e$t$(j);:gosub8400:
ifj<4goto9230
 9300 print"<clr><c-d>i can also go forward if you type f.
 9310 print"<c-d>when i reach the edge of the screen,
 9320 print"<c-d>i just stop.  as i move around the
 9330 print"<c-d>screen, i erase what was there.
 9340 print"<c-d>try it now.  ";:x=13:y=9:j=1:d=j:x$="neswfq
 9350 printe$t$(d);:gosub8400:ifj=4thengosub8200:goto9350
 9360 ifj<4thend=j:goto9350
 9400 print"<clr><c-d>with some special help, i can remember
 9410 print"<c-d>how you tell me to move.  what i learn
 9420 print"<c-d>is called a program.  i need two new
 9430 print"<c-d>commands: c to clear the screen and
 9440 print"<c-d>          h to send me home (top left).
 9450 print"<c-d>when you hit a key, i will run a
 9460 print"<c-d>program called 'square'.
 9470 p$="c"+p$(1,0):print"<c-d><c-d>'square'="p$:gosub8000
 9500 gosub30:print"<home><c-d><c-d><c-d><c-d><c-d>the program cal
led 'square' said
 9510 print:print"   "p$:print"<c-d>c clear the screen
 9520 print"<c-d>e turn east
 9530 print"<c-d>ffff take four steps forward
 9540 print"<c-d>sffffwffffnffff draw the right, bottom,
 9550 print"<c-d>   and left sides of the square.
 9560 print"<c-d>do it again";:gosub8050:ifj=0goto9500
 9600 print"<clr><c-d>the reason i could show you 'square'
 9610 print"<c-d>again is that i remembered it.
 9620 print"<c-d>i learned its name when i saw it in
 9630 print"<c-d>the single quotes (') and learned what
 9640 print"<c-d>it said to do when i saw the = sign.
 9650 print"<c-d>so, saying 'square'=
 9660 print"<c-d>told me to learn the program and to
 9670 print"<c-d>call it 'square'.  notice that 'square'
 9680 print"<c-d>did not need a q for quit.  when a
 9690 print"<c-d>program is over, it quits by itself.":gosub8000
 9700 print"<clr><c-d>i can remember up to ten different
 9710 print"<c-d>programs as long as they have different
 9720 print"<c-d>names.  one program can use another
 9730 print"<c-d>by putting its name on the right of the
 9740 print"<c-d>= sign.  let's have another program:
 9750 sq$=p$(1,0):st$=p$(1,1):print"<c-d>'step'=efsf
 9760 print"<c-d>'step' tells me to go across one and
 9770 print"<c-d>down one, like going down one step of
 9780 print"<c-d>a staircase.  i can just step down the
 9790 print"<c-d>screen - like this:":gosub8000
 9800 p$="c"+st$+st$+st$+st$+st$+st$+st$+st$+st$+st$:gosub8040
 9820 print"<clr><c-d>that program used a c for clear, then
 9830 print"<c-d>ten steps by using 'step' ten times.
 9840 print"<c-d>if we take the c for clear out of
 9850 print"<c-d>'square', we can put it together with
 9860 print"<c-d>'step' and call it '2 step'=
 9870 p$="c"+p$(1,2)
 9880 print"<c-d>c'square''step''square''step''square'
 9890 print"<c-d>when you're ready to watch it,":gosub8000:gosub80
40
 9900 print"<clr><c-d>'2 step' is the third program i have
 9910 print"<c-d>learned already.  you may now tell
 9920 print"<c-d>me to run one i already know by typing
 9930 print"<c-d>its name <c/r>, or name a new one using
 9940 print"<c-d>the = sign.  don't forget to put the
 9950 print"<c-d>program name in apostrophes - 'step'.
 9960 gosub8000:goto100
 9990 fori=0to999:nexti:return:rem delay
 10000 data"QI<c-l><c-l><c-l>U<c-u>W<c-d><c-l>
 10010 data"QW<c-l><c-l><c-u>I<c-d><c-d><c-l>K<c-u><c-l>
 10020 data"QK<c-l><c-l><c-l>J<c-d>W<c-u><c-l>
```

```
10030 data"Q<c-l><c-l>W<c-u>U<c-d><c-d><c-l>J<c-u><c-l>
10050 data"square",effffsffffwffffnffff,"step",efsf,"2 step
10060 dataeffffsffffwffffnffffefsfeffffsffffwffffnffffefsfeffffsf
fffwffffnffff
```

CHECKSUM: 21

```
Line-number references

 100   190   510  1190  2010  2020  9090  9960
1000   140
1190  1110  1120  1140
 150   165   180   370  1150
 190   360
  20    20  8010  8100
2000   150
  22    22  8320
  30  8040  9500
 300   160
 330  1030
  50    50  8400
 500   170   320   350  1020
  52    52
8000   190   510  1190  9180  9470  9690  9790  9890  9960
8040  2010  8040  9800  9890
8050  8040  9090  9560
8100  8120
8110    32
8200    36  9350
8300   130
8310  8310
8320  8330  8350
8400  8410  9230  9350
9000    10
9230  9230
9350  9350  9360
9500  9560
9990  8040  9120  9130  9140  9150  9160
```

```
Variable references

   a   130  1150  2010  2020
   c    24   160   160   165   170   180  8320  8330  8340  8340
  c$    20    20    22    22    24    32    50    50    52    52
  c$   180   180   180  8110  8310  8310  8330  8340  8340  8400
   d    38    39  8200  8210  8220  8230  9340  9350  9360
  d$  8240  9030
  e$    39  8200  9030  9130  9140  9150  9230  9350
   i   310   310   310   310   340   340   340   340  1010  1010
   i  1010  1010  1130  1130  1130  1130  1130  1130  8110  8110
   i  8110  8110  8120  8400  8400  8400  8400  8410  9020  9020
   i  9040  9040  9040  9050  9050  9990  9990
   j    32    34    34    34    36    38    38   310   310   320
   j   330   330   340   340   350   360   370  1010  1010  1020
   j  1030  1110  1120  1130  1130  1140  1150  8040  8110  8400
   j  9090  9220  9230  9230  9230  9340  9340  9350  9360  9360
   j  9560
   k    32    32    32    39
 k%(   170  9050  9050
  p$    30    32    32   130   180   180   180   360   370   370
  p$  2010  2010  2020  9470  9470  9510  9800  9870
 p$(   340   360   370  1130  1130  1130  2020  9040  9040  9040
 p$(  9470  9750  9750  9870
sq$  9750
st$  9750  9800  9800  9800  9800  9800  9800  9800  9800  9800
st$  9800
t$(    39  9020  9020  9100  9120  9130  9140  9150  9230  9350
  x    34  8210  8210  8210  8230  8230  8230  8240  9340
 x$    30   140   150   160   165   165   180   180   310   310
 x$   330   330   330  1010  1010  1030  1110  1120  1120  1120
 x$  8050  8110  8110  8300  8330  8330  8330  8330  8340  8340
 x$  8400  8400  9050  9050  9050  9220  9230  9340
  y    34  8200  8200  8200  8220  8220  8220  8240  9340
 y$   330   340   370  1110  1110  1130  1130  1150
```

5.10 DOMINOES

Some years ago, it was surprising to note that no one had put the game of dominoes on a computer. A little analysis showed why it hadn't been done, some thinking showed a solution to the problem, and a little work put a predecessor of the program on the sample disk on the market.

Dominoes is played with "bones" that show two patterns of dots. Each bone is unique, and each combination of zero through six dots on each side is on one bone. The total number of bones is therefore 28. To display a bone takes at least 5 × 8 pixels. Theoretically, all plays could go on one side (or even one quadrant) of the screen—there just isn't room to show them decently with only 1000 pixels.

In practice, the only important things about the bones already played in the block game are the two active ends. Thus, we can make a display with those ends shown in the center, and use just numbers to represent those in the player's hand and those already played by both sides. Each bone displayed takes two pixels (width) by four (height) for legibility. We can put up to 20 bones on a row, so we need two rows to display those already played. Each player starts with 7 bones, leaving 14 in the bone yard. In the worst case, the computer could have all 7 bones with 6's on them in its starting hand. Then the player would have to draw all 14 before passing, making a total of 21, too many to show on one row. For that very improbable case or the one requiring 20 bones in the player's hand, we might want to have an extra row on the display. On the other hand, the chance that that condition will ever arise is so low that it could be ignored.

The DOMINOES program is fully operational and plays a pretty good game. Its best play has three levels of logic: play a doublet if possible; play a tile that has a successor if possible; pick at random from those that can be played. Doublets are handicaps since they are less easily played than bones with different numbers on the two sides. Therefore, it's harder to play your last bone if it's a doublet, and your chance of winning is reduced. (Double zero is an exception, but the program doesn't recognize that.) There are many other strategies that can be added to the system, usually taking extra time.

A quick review of the listing will show that DOMINOES falls short of "advanced software" as developed in this text. It isn't interoperable and its speed could be improved. Before you make changes, make sure that you really know the rules by checking a copy of Hoyle. You can get fancy by using one of the variants of the game, but just improving the basic block game will be useful.

```
dominoes                                        Non-Professional

 5 gosub15000:rem m.richter 90064 12/78
 10 pw=0:tw=0:ty=0
 20 poke59468,12:i=rnd(-ti):dimd$(27),h$(1,20),pl(6),h(20)
 40 lu$="<home><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d><c-d>
<c-d><c-d><c-d><c-d><c-d><c-d><c-d>":lo$=lu$+"
                     <c-u>"
 100 pl=0:print"<clr>"spc(15)"dominoes<c-d><c-d><c-d><c-d><c-d>
<c-d><c-d><c-d><c-d><c-d><c-d>
 110 j=0:k=0:fori=0to27:d$(i)=mid$(str$(j),2)+mid$(str$(k),2):
gosub9500
 120 k=k+1:ifk>jthenk=0:j=j+1
 130 nexti:fori=0to26:j=i+(28-i)*rnd(1):x$=d$(i):d$(i)=d$(j):d$(j)
=x$:gosub9500
 140 nexti:forj=0to6:fori=0to1:h$(i,j)=d$(7*i+j):nexti:gosub9500:
nextj:sp=14
 150 forj=7to20:fori=0to1:h$(i,j)="":nexti:gosub9500:nextj
 160 k0$="d":k1$="<rvs>d<off>raw":k2$="draws a bone
 200 l$="":forj=0to1:fori=0to6:x$=left$(h$(j,i),1):gosub9500
 210 ifx$=right$(h$(j,i),1)andx$>l$thenl$=x$:p=j:k=i
 220 nexti:nextj:ifl$=""goto100
 230 print"<home><c-d><c-d>p<c-d><c-l>l<c-d><c-l>a<c-d><c-l>y<c-d>
<c-l>e<c-d><c-l>d<c-d><c-d>
 240 printspc(8)"U⁚⁚⁚⁚"""⁚⁚⁚⁚I   bone count
 250 printspc(8)"|    <rvs>   <off>    |   DDDDDDDDDD
 260 printspc(8)"| V  <rvs>   <off>  V |   yard
 270 printspc(8)"|    <rvs>   <off>    |   you
 280 printspc(8)"J⁚⁚⁚⁚<rvs>999<off>⁚⁚⁚⁚K   pet
 290 gosub2000:gosub7000
 295 printlo$:print"the largest doublet is ";l$l$:r$=l$:gosub1980:
goto1000
 300 p=0:printlo$:print"which bone (or "k1$")? ";
 302 gosub4000:x$=c$:ifx$=k0$thenprintx$:goto8000
 305 ifx$<"0"orx$>"6"goto302
 307 printx$;:
 310 gosub4010:y$=c$:ify$<"0"ory$>"6"goto310
 315 printy$:x$=x$+y$:y$=y$+left$(x$,1)
 320 fork=0to20:ifh$(0,k)<>x$andh$(0,k)<>y$thennextk:goto9000
 330 rf=0:ifpl=0thenl$=right$(x$,1):r$=left$(x$,1):lf=1:goto1000
 340 lf=0:ifl$=left$(x$,1)orl$=right$(x$,1)thenlf=1
 350 ifr$=left$(x$,1)orr$=right$(x$,1)thenrf=1
 360 iflf+rf=0goto9000
 370 iflf*rf=0thenonlf+1goto500,550
 375 ifl$=r$orh$(0,1)=""goto450
 380 printlo$:printx$" - left or right? ";
 390 getc$:ifc$<>"l"andc$<>"r"goto390
 400 printc$:ifc$="l"goto450
 410 ifr$=right$(x$,1)thenr$=left$(x$,1):goto1000
 420 r$=right$(x$,1):goto1000
 450 ifl$=right$(x$,1)thenl$=left$(x$,1):goto1000
 460 l$=right$(x$,1):goto1000
 500 ifr$=right$(x$,1)thenr$=left$(x$,1):goto1000
 510 r$=right$(x$,1):goto1000
 550 ifl$=right$(x$,1)thenl$=left$(x$,1):goto1000
 560 l$=right$(x$,1):goto1000
 1000 rem play the bone
 1005 l%=0:y$=h$(p,k):fori=kto19:h$(p,i)=h$(p,i+1):nexti:h$(p,i)="
":gosub7000
 1010 gosub2000:x$=y$:print"<home><c-d>":gosub5000:pl=pl+1
 1020 x=pl:ifx>18thenx=x-18:print"<c-d><c-d><c-d>
 1030 printspc(1+2*x)x$
 1100 x$=l$:gosub5900:printleft$(lo$,12)spc(9)x$:x$=r$:gosub5900
 1110 print"<c-u><c-u><c-u>"spc(17)x$:ifh$(0,0)=""orh$(1,0)=""goto
8500
 1150 ifpgoto300
 1200 rem pet's play
 1210 p=1:onalgoto1450,1350
 1220 fori=0to6:pl(i)=0:nexti
 1230 fork=0to20:ifh$(1,k)=""goto1250
 1240 gosub9900:x=val(x$):pl(x)=pl(x)+1:x=val(y$):pl(y)=pl(y)+1:
nextk
 1250 rem playable doublet with successor
 1260 fork=0to20:ifh$(1,k)=""goto1300
 1270 gosub9900:if(x$=y$)and(x$=r$orx$=l$)and(pl(val(x$))>2)goto15
00
 1280 nextk
 1300 rem playable bone with successor
 1310 fork=0to20:ifh$(1,k)=""goto1350
 1320 gosub9900:if((x$=l$)or(x$=r$))andpl(val(x$))>1goto1500
 1330 if((y$=l$)or(y$=r$))andpl(val(y$))>1goto1500
 1340 nextk
 1350 rem playable doublet
 1360 fork=0to20:ifh$(1,k)=""goto1400
 1370 gosub9900:ifx$=y$and(x$=l$orx$=r$)goto1500
 1380 nextk
 1400 rem still more algorithms here
 1450 rem playable bone
 1460 fork=0to20:gosub9900:ifl$=x$orr$=x$orl$=y$orr$=y$goto1500
 1470 nextk:printlo$:print"pet "k2$:gosub1980:goto8000:rem pet dra
ws or passes
 1500 rem wrapup pet's play
 1510 printlo$:print"pet plays "x$y$:gosub1980
 1520 x=val(x$):pl(x)=pl(x)-1:x=val(y$):pl(x)=pl(x)-1
 1550 ifl$=x$thenl$=y$:goto1000
 1560 ifl$=y$thenl$=x$:goto1000
 1570 ifr$=x$thenr$=y$:goto1000
 1580 r$=x$:goto1000
 1980 ti$="000000
 1990 ifti<60goto1990
 1995 return
 2000 rem put up count
 2120 printleft$(lo$,13)spc(30)28-sp"<c-l> "
 2130 fori=0to20:ifh$(0,i)>""thennexti
 2150 printspc(30)i"<c-l> ":fori=0to19:ifh$(1,i)>""thennexti
 2170 printspc(30)i"<c-l> ":return
 4000 getc$:ifc$>""goto4000
 4010 getc$:ifc$=""goto4010
 4020 return
 5000 x$=left$(x$,1)+"<c-l><c-d>:<c-l><c-d>"+right$(x$,1):return
 5900 rem get dot format
 5990 on1+val(x$)goto6000,6010,6020,6030,6040,6050,6060
 6000 x$="   <c-d><c-l><c-l><c-l>   <c-d><c-l><c-l><c-l>   ":
return
 6010 x$="   <c-d><c-l><c-l><c-l> Q <c-d><c-l><c-l><c-l>   ":
return
 6020 x$="Q  <c-d><c-l><c-l><c-l>   <c-d><c-l><c-l><c-l>  Q":
return
 6030 x$="Q  <c-d><c-l><c-l><c-l> Q <c-d><c-l><c-l><c-l>  Q":
return
 6040 x$="Q Q<c-d><c-l><c-l><c-l>   <c-d><c-l><c-l><c-l>Q Q":
return
 6050 x$="Q Q<c-d><c-l><c-l><c-l> Q <c-d><c-l><c-l><c-l>Q Q":
return
 6060 x$="Q Q<c-d><c-l><c-l><c-l>Q Q<c-d><c-l><c-l><c-l>Q Q":
return
 7000 rem print player's dominoes
 7010 printlu$"<c-d><c-d><c-d>":fori=0to19:ifh$(0,i)>""goto7100
 7030 ifi<20thenprint" <c-l><c-d> <c-l><c-d> "
 7040 i=0:return
 7100 x$=h$(0,i):gosub5000:printx$;:ifi<19thenprint"<c-u> <c-u>";
 7110 nexti:return
 8000 rem pet draws or passes
 8010 ifsp=28goto8100
 8020 fori=0to20:ifh$(p,i)>""thennexti
 8025 printleft$(lu$,13)
 8030 ifsp>26thenprint"<rvs> yard":print"<rvs>empty":k0$="p":k1$="
<rvs>p<off>ass":k2$="passes
 8040 h$(p,i)=d$(sp):sp=sp+1:gosub7000:gosub2000:onp+1goto300,1200
 8100 rem pass logic
 8110 ifl%goto8500
 8120 l%=1
 8200 onp+1goto1200,300
 8500 s0=0:fori=0to19:x$=h$(0,i)
 8510 ifx$>""thens0=s0+val(left$(x$,1))+val(right$(x$,1)):nexti
 8520 s1=0:fori=0to19:x$=h$(1,i)
 8530 ifx$>""thens1=s1+val(left$(x$,1))+val(right$(x$,1)):nexti
 8540 ifs0>s1thenx$="<rvs> pet wins  ":pw=pw+1
 8550 ifs1>s0thenx$="<rvs> you win   "
 8560 ifs0=s1thenx$="<rvs>it's a draw":ty=ty+1
 8570 t0=t0+s0:t1=t1+s1:tw=tw+1:gosub2000
 8600 printlo$:print"<rvs>"x$"<off> score: you"s0", pet"s1:gosub19
80
 8605 print"___________":print"<rvs> another?  "
 8610 gosub4000:ifc$="y"goto100
 8620 ifc$<>"n"goto8610
 8800 rem wrapup
 8820 print"<clr><c-d><c-d>of the"tw"games, you won"tw-ty-pw"pet w
on"pw
 8830 iftythenprint"and we drew"ty
 8840 print"<c-d>the totals were: you"t0", pet"t1
 8850 print"<c-d><c-d>to play again, enter 'run' when you're":end
 9000 printlo$:print"try again: ";:goto302
 9500 print"<c-u>"spc(14);:ifuthenprint" ";
 9510 print"shuffling ":u=1-u:return
 9900 x$=h$(1,k):y$=right$(x$,1):x$=left$(x$,1):return
 15000 poke59468,12:print"<clr><c-d>","<rvs>   dominoes   <off>":
print,"<c-d><rvs> instructions <off>
 15005 print"<c-d><c-d>to play a bone type in the two numbers.
 15010 print"<c-d><c-d>to draw a bone type "chr$(34)"d"chr$(34)
 15020 print"<c-d><c-d>note: it is not necessary to press
 15025 printchr$(34)"return"chr$(34)".
 15030 print"<c-d><c-d><c-d>press a key to continue
 15040 gett$:ift$=""goto15040
 16000 print"<clr><c-d><c-d><c-d>":print"pet has three levels of p
lay:
 16010 print"<c-d>1. easy: you can win two games of three
 16020 print"<c-d>2. slightly harder: maybe 60/40 for you
 16030 print"<c-d>3. tough: nearly equals best play
 16040 print"<c-d><c-d>which do you want?
 16050 gosub4000:ifc$<"1"orc$>"3"goto16050
 16060 al=val(c$):return
 20100 printtab(9)"V";:printtab(29)"V":return
```
CHECKSUM: 60

```
Line-number references

   100   220  8610
  1000   295   330   410   420   450   460   500   510   550   560
  1000  1550  1560  1570  1580
  1200  8040  8200
  1250  1230
  1300  1260
  1350  1210  1310
  1400  1360
  1450  1210
  1500  1270  1320  1330  1370  1460
 15000     5
 15040 15040
 16050 16050
  1980   295  1470  1510  8600
  1990  1990
  2000   290  1010  8040  8570
   300  1150  8040  8200
   302   305  9000
   310   310
   390   390
  4000   302  4000  8610 16050
  4010   310  4010
   450   375   400
   500   370
  5000  1010  7100
   550   370
  5900  1100  1100
  6000  5990
  6010  5990
  6020  5990
  6030  5990
  6040  5990
  6050  5990
  6060  5990
  7000   290  1005  8040
  7100  7010
  8000   302  1470
  8100  8010
  8500  1110  8110
  8610  8620
  9000   320   360
  9500   110   130   140   150   200
  9900  1240  1270  1320  1370  1460
```

```
Variable references

   al  1210 16060
   c$   302   310   390   390   390   400   400  4000  4000  4010
   c$  4010  8610  8620 16050 16050 16060
  d$(    20   110   130   130   130   130   140  8040
  h$(    20   140   150   200   210   320   320   375  1005  1005
  h$(  1005  1005  1110  1110  1230  1260  1310  1360  2130  2150
  h$(  7010  7100  8020  8040  8500  8520  9900
   h(    20
    i    20   110   110   130   130   130   130   130   130   140
    i   140   140   140   140   150   150   150   200   200   210
    i   210   220  1005  1005  1005  1005  1005  1220  1220  1220
    i  2130  2130  2130  2150  2150  2150  2150  2170  7010  7010
    i  7030  7040  7100  7100  7110  8020  8020  8020  8040  8500
    i  8500  8510  8520  8520  8530
    j   110   110   120   120   120   130   130   130   140   140
    j   140   140   150   150   150   200   200   210   210   220
    k   110   110   120   120   120   120   210   320   320   320
    k   320  1005  1005  1230  1230  1240  1260  1260  1280  1310
    k  1310  1340  1360  1360  1380  1460  1470  9900
  k0$   160   302  8030
  k1$   160   300  8030
  k2$   160  1470  8030
   l$   200   210   210   220   295   295   295   330   340   340
   l$   375   450   450   460   550   550   560  1100  1270  1320
   l$  1330  1370  1460  1460  1550  1550  1560  1560
   l%  1005  8110  8120
   lf   330   340   340   360   370   370
  lo$    40   295   300   380  1100  1470  1510  2120  8600  9000
  lu$    40    40  7010  8025
    p   210   300  1005  1005  1005  1005  1150  1210  8020  8040
    p  8040  8200
   pl   100   330  1010  1010  1020
  pl(    20  1220  1240  1240  1240  1240  1270  1320  1330  1520
  pl(  1520  1520  1520
   pw    10  8540  8540  8820
   r$   295   330   350   350   375   410   410   420   500   500
   r$   510  1100  1270  1320  1330  1370  1460  1460  1570  1570
   r$  1580
   rf   330   350   360   370
   s0  8500  8510  8510  8540  8550  8560  8570  8600
   s1  8520  8530  8530  8540  8550  8560  8570
   sp   140  2120  8010  8030  8040  8040  8040
   t$ 15040 15040
   t0  8570  8570  8840
   t1  8570  8570
   ti    20  1990
  ti$  1980
   tw    10  8570  8570  8820
   ty    10  8560  8560  8820  8830  8830
    u  9500  9510  9510
    x  1020  1020  1020  1020  1030  1240  1240  1240  1240  1520
    x  1520  1520  1520  1520  1520
   x$   130   130   200   210   210   210   302   302   302   305
   x$   305   307   315   315   315   320   330   330   340   340
   x$   350   350   380   410   410   420   450   450   460   500
   x$   500   510   550   550   560  1010  1030  1100  1100  1100
   x$  1110  1240  1270  1270  1270  1270  1320  1320  1320  1370
   x$  1370  1370  1460  1460  1510  1520  1550  1560  1570  1580
```

```
x$  5000  5000  5000  5990  6000  6010  6020  6030  6040  6050
x$  6060  7100  7100  8500  8510  8510  8510  8520  8530  8530
x$  8530  8540  8550  8560  8600  9900  9900  9900  9900
 y  1240  1240
y$   310   310   310   315   315   315   315   320  1005  1010
y$  1240  1270  1330  1330  1330  1370  1460  1460  1510  1520
y$  1550  1560  1570  9900
```

6 Pursuing a Project

There is no formula for writing advanced software. There is no pattern to the way projects come to you. This chapter is devoted to one scenario, where you are designing and building an inventory system for a specific customer. We will walk all the way through the process, from the first phone call to the first request for expansion. Notice that we're taking the easy path—a project requested by a customer. We're even simplifying that, detailing only the easiest of the three programs in the system. In partial atonement, the rest of this section looks briefly at a project that you start on your own.

You're an amateur fly-tier, with a passion for the classic patterns and their modern variations. (Not *that* kind of fly—these are fishing lures, and can represent high art in the opinion of a devotee.) Your collection includes samples and pictures. You want to catalogue the collection, to develop a taxonomy based on size, shape, material, color, and other features. Then you will add measures of performance, and eventually develop a science of fly-tying to complement the art. With sudden insight, you decide to put your computer on the job with you. You'll write a catalogue system, FLYTIE, and revolutionize the field. For this, you will be customer, user, designer, coder, tester, and everything else.

There are two interesting possibilities here: you will be the only user of your product; or you will be the pioneer in a new field of thousands (dozens?). If no one else will ever use your system, then write what you will and as you will. If it works for you, good enough! But maybe there will be a market; maybe the fly-tying field will be revolutionized by your product. How do you handle that prospect?

For fly-tying alone, there is little to add to the material in the rest of this chapter. You will have to play all the roles and pretend to be a computer novice when acting as the customer or user. The larger problem comes when you think about what else the system should do. At the least, remember that other fly-tiers will want features that you don't. How do you extend your ideas to address the broadest potential market? Concentrate on the most general statement of the problem that satisfies all of your needs. (Don't lose sight of your basic requirement to catalogue your own collection; just try to build on it.)

Can the system be extended still further? Can the same techniques be applied in biology or geology? Is there a still more general concept of cataloguing that opens the market still further? If the answers suggest a broader kind of system than you planned, there are several ways to get there from your original idea. One of the best approaches is gradual; trying for the ultimate system first can leave you dissatisfied—even disgusted—with beautiful parts of a package that doesn't work.

Start by building something that directly addresses your own fly-tying needs. Get it up and working first to put your house in order. Then take what you learned and what you have working to investigate extensions for other applications. Can you add more fields? Can you tokenize more information to save space? Would a relative file be better than sequential or vice versa? Sketch alternative designs until you find something that looks to be marketable. Now, follow the methods of advanced software to build the new system. How do you test it? Use your FLYTIE catalogue. Move the files that work into the new system, and verify that it does everything that the specialized program did. And don't forget to document. If anyone else is ever to use your product, the rules must be legible.

6.1 The first call

"Are you the computer whiz they told me about?" Your first contact with a customer may start that way, inspired by a recommendation from "them" to her. Who gave that recommendation? A satisfied customer, someone who heard you lecture, or a beleaguered dealer may have been the source. Whoever it was, she thinks you're the miracle worker who will move her business into the computer age in three days for $49.95 (plus tax). You have to persuade her that she found the right person at the same time that you tell her that your stock of miracles has run out.

The customer will want to dump her problem on you over the phone. Don't let her. Take enough information to find out whether you have a chance of solving the problem, then ask for a face-to-face meeting to get more. Before making that request, use the call to scale the problem, calibrate the customer, estimate the time frame, and comfort the customer.

Does the problem belong on a pocket calculator or a mainframe? If it isn't in the range that you handle comfortably, pass the job on to someone else. A microcomputer won't process five thousand transactions a day; if that's what she needs, recommend someone who programs mainframes. A distributed data base won't sit comfortably on Commodore, and an attempt to shoehorn it in won't pay off.

Five minutes on the phone will give you a good idea of your customer's competence. The worst customers have only enough knowledge to think they know computers. The best are those who admit to complete ignorance or who know enough to recognize your expertise and leave you alone. The customer who thinks the computer will do everything needs to be disabused. The one who tells you exactly what to do and how should be invited to do it herself. Fortunately, most customers will work with you for their benefit—just make sure you show them where that advantage lies. Remember that software experts are in short supply, so she has to sell her problem to you as hard as you sell your services to her.

There is no chance that a call in March will get an accounting system on line in time for April's taxes. Make sure that the amount of work is consistent with the calendar and your availability. Some problems can be worked in stages: an initial capability in a few months; bells, whistles and special reports months later. Your hardest work—thinking—must be completed before you do any coding, so don't fool around. Building half the system may take 80% of the effort of building it all, and at least 80% of the time.

Not all first calls come in panic. Just most of them, and they need calming and assurance that you're the right person to do the job, that you can learn her needs and satisfy them, and that you can do so fully, quickly and economically. She needs calming first, solutions second. If you take a strong, positive approach in the first call, she'll assume that you will work her problem strongly and positively. And that's what she needs to believe. Remember that your customer is over her head. She's calling you the same way that she would her physician or her architect. You are a professional and must sound like one.

If you haven't screened out the job or the customer, the final point of the first call is to set up a meeting. The best place is on her turf, not yours. She will be comfortable and in control, with all information at hand. You won't be tempted to fall into jargon, or to jump to the machine to demonstrate your insight. (A flashy show on the computer is likely to persuade her that she's incompetent to deal with it—or you.) Remember to speak English on the phone and at your meeting; she isn't fluent in computer or BASIC, and won't hire someone she can't talk to. When you schedule the appointment, make sure that you'll each have at least an hour. Set it up as soon as it is convenient, but don't catch her panic. It will take weeks or months to solve her problem; a little leisure in defining it can only help.

6.2 First meeting

Your first meeting with your potential customer is a job interview. She must find out whether you're qualified; you need to learn what job you're trying for. For a conventional interview, you prepare a resume. For this one, assemble your one-page product descriptions. Put on a cover letter that says: here's a sample of what I have already done for others. Your job is different, but it can be based on my solid base of experience. The descriptions will demonstrate your qualifications. Pick the examples based on the new job, and put them in order of decreasing relevance to it. Drop the package off before the meeting if time permits. By reading them first, she will not only know that you have the technical problem under control but also that you're prepared to talk in English, not computer.

When you go into the meeting, recognize its purposes. The customer is to be assured of your competence, but that requires no special action. Asking the right questions and handling her answers well will do that automatically. Keep her at ease, and try to be comfortable yourself. When you leave the meeting, you will write up the one-page customer description of her problem. So you will have to learn enough to do that during the conversation.

The Widget Company sells 100 different products from 30 warehouses. They need to track their inventory, keep stocks above a target level in each item, and distribute products to their customers as efficiently as possible. Each warehouse holds enough for a month of normal business, but their paper shuffling has gotten out of hand. They log each item moved, but can't process the paper fast enough to know what's in stock where—except by going back to count it. There may be a problem with pilfering, but they aren't sure. How do you help?

Widget Company needs inventory control. They need a lot of input stations (30), but don't require real-time operations. So each station can stand alone, and can talk to inventory control through floppy disks, telephone lines, or even cassettes. What operations should go on at a station? Ask, don't guess. How about instant local inventory? (Widget's customer calls for 25 pieces in size 17; are they in stock? If so, an instant answer is available. Can this office call central to find out where the customer can get them? Does it have to be able to answer without that call? How many times a day is that kind of question asked? How often will the request have to be satisfied outside that warehouse? How quickly are responses needed? Is the cost of a disk acceptable if all it provides is that sort of instant inventory? Would a printout do as well?)

As you talk about your customer's needs, you start to postulate a design. That leads to more specific questions. Can she live with a limit of 200 items? Can we always enter a new item from the central office, or must a station be able to add to the list of items it carries? Can we be sure that there will never be more than 31 stations? 63? Are all stations equivalent? Those questions are directed at the main purpose of the system.

You also check out the other things the system will have to do—even if they are to be developed much later. When an item is sold, should the station log the customer? Is there a fixed list of customers? What about walk-in business? Is there any problem if the customer list is maintained only at the home office? How about suppliers? Must you distinguish equivalent parts by supplier? Can we assign our own part numbers? If we do, what restrictions apply? How about the station printing the invoice for the order? Figuring the price? Recording payment on a walk-in? Account balance? Check balance against that customer's limit?

When you leave that first meeting, you will have 90% of the information you need to design the system. You will have two choices for the missing 10%: call and ask, or guess. The calls will be difficult and should be few in number and directly to the point. When you know what you don't know, plan a call. List your questions. For best results, have multiple-choice answers ready. Before you leave the meeting, find out when you can telephone, and set up a tentative appointment to present your solution. Regard that appointment as an absolute commitment on your part, which the customer can change at a moment's notice. Call before that appointment to confirm it. Again, plan that the second session is at the customer's office. That's where the answers you need will be found; that's where you can look at existing paper processes.

Make sure that your customer knows that you understand her problems. One that you should raise (in order to put it aside) is transition to the new system. The customer is doing something already, and it has worked until now. You will not throw all that good stuff away just because it doesn't fit the computer. Invite them to suggest additional testing, and ask the user to participate. If he's willing, let him join you for the testing itself. Again, the customer will have adopted your program as her own even before delivery.

6.3 Structuring the system

The week or so following the first meeting with your customer is your time for thinking. That's when you lay out your file and system structures, devise the way she will get the functions and information she needs from the product, size the work involved for schedule and cost, and write up the customer documentation. You're a long way from coding, and your computer at this stage is either idle or a word processor.

Widget Company's products have names as long as 30 characters; 45 of them will go in a string array of 1350 characters total, and we can afford to keep it in memory. If they average 20, even 200 different names will fit. There are 30 warehouses with 100 items each, so we'd need over 20K to hold the inventory in floating point. Integers will do the job and will take only about 6K, so the inventory could go into memory, too. If they grow to 60 stations and 150 items, it might be tight; let's plan on using the inventory file from disk and leave room for expansion. The company needs the customer information as well, but we'll use a standard mailing list for that. We'll call each line item a transaction. It is characterized by its customer, part number, quantity, price per item, discount and tax rates, and mode of payment. Each station will need a file of all customers, tax and discount rates, authorized signatures, available credit, and comments and special conditions. The central office will do monthly reporting and billing from the transaction logs. You have to check on whether they want a general ledger, and if so whether it has to be linked into the inventory system automatically.

Each station will keep its own inventory current by subtracting each sale and adding each delivery when it occurs. (Deliveries are transactions, too. Remember them in designing the log.) Initialize the system with the date; item and customer names are read in from the current disk. The date is converted to a number, and the day's log file is created. Check with the customer about whether a review of inventory should be automatic at initialization—the system could flag shortages before the day's business starts. As a customer comes to the station to check out, the transaction is logged and inventory decremented. A customer's call may require an inventory check. Naming an item can be by complete part number (easy on the machine) or by name (easy on the clerk). By name, we should use keyword search; find a three-inch quarter-twenty bolt by looking at all bolts if you want, then accessing by part number. Price structure is part of the item information; since it varies by quantity, we need, say, five prices per item. That file won't fit in memory even with only 100 items. One more job at the station would be to check on whether another station has parts not available here; that software is simple enough to be added later and requires no change of file structure. So, we'll leave that for add-on after the system is running.

The central system will take the logs in by disk and add fields for date and station ID before recording them. It needs a dual drive for that transcription. Daily activity is to integrate yesterday's transactions into the history, flag special conditions (e.g., account near limit, parts requiring manufacture or reorder), and create new files for the stations. Problems: two trips to each station per day, one to take out the morning's disk, the other to pick up the one with the transactions. Solution may be to use telephone (modem) communications after the system is set up. Recognize the problem now, make sure there is a solution that won't break the system, then continue with primary design.

The central system needs dual disk and a quality printer, which means an IEEE system at $3,000–5,000. Each station can run with a 64 (maybe even a VIC) with a 1541 and a 1525 (for on-the-spot invoicing). Say $1500 for each of the 30 stations. So, the customer is looking at $50,000 in hardware all told. She'll probably balk at that total and start with fewer stations. The five busiest stations will get the full treatment; the others will work on paper at first. The bill will then be around $12,500. If the software runs another $2,500, we mean about a 100-hour job. Allowing half that time for testing and data entry, we need to be sure that $1,250 or so is a fair price for your design and coding.

Two other items need to be considered at this stage, both dealing with interfacing the system. Firing up the first time will require entering a complete inventory. Thereafter, occasional inventories will verify the history, check for pilferage, and keep the accountants happy. Inventories require facilities for data entry and reports of discrepancies. The initial system will require manual entry for warehouses without stations; that capability will fit in with file editing and transaction correction. Until clerks become perfect, errors will occur; the system has to be able to tolerate and to correct them. We'll need a system to identify transactions that are current and to purge those that are not. Someday we'll want programs to analyze trends and aid long-range planning, so purged logs shouldn't be thrown away. We'll keep the old disks with their logs, and we'll maintain a sequential file of purges. Between them, we can construct a complete audit trail.

The other set of functions of the central system we can think of as being done monthly. (The actual interval is whenever the user wants to do them.) That run reports activity by station and by customer; its primary purpose is to generate bills for the current accounts. A later generation of system may well provide a cover letter, but for now we'll just summarize monthly activity. One line per transaction per account. Then part of the daily activity has to be logging payments that come in. (Check need for distinct status flags: received, cleared. When should we credit the checks? Need we compute interest on outstanding balance?) Put in a mechanism for interest on daily balance after a fixed delay. Initial interest rate will be set to 0; to set a rate, change the line and save the program again. It makes no sense to ask the question repeatedly. More pressing is the need to flag each billed transaction so it is not recomputed.

We'll log for each customer the date billed and amount due (for next time) and tick off everything we billed. We can purge the old transactions from the system (put them into the historical file) when the bills are prepared or when they're paid. Nominally, let's do it at preparation, and turn back to paper copies (or historical search) if questions arise after billing. Those functions could be in the same program as the daily central functions, but they are really different and probably need a bookkeeper to run them instead of the central clerk. Both programs end with long print runs, so have different kinds of wrapup.

The system will need three distinct programs: warehouse (interoperable 64/central), daily (central), and monthly (central). For backup, we need a second IEEE system (we can do without a second letter-quality printer). It can back up a warehouse station as well as the central one since it can run station software. Total hardware cost is about $15K. We'll go for $3K for the baseline software. If that's too big a bite at one time, we can back off to a single, central station without warehouse software at $5K hardware, $2K software on a two-month development schedule. Knowing our file structures, we can add the warehouse hardware and software at any time. (Most of the $1K saved in software comes from not having to test the interface, not from skipping the warehouse program.)

The central station will serve as a warehouse station during the day; it will run the same program that goes into their 64's. (It should probably have a dot-matrix printer in addition to its 8300; otherwise, the programs might not be identical.) A new customer will be added on second shift, when the disks are updated and the item information (prices, etc.) is modified. With that, we have pretty much pinned down the architecture we're proposing and are ready to write the customer description.

6.4 Inventory control (customer description)

The Inventory Control System (ICS) is a package of custom software and Commodore hardware that manages the inventory and accounts receivable for a coordinated warehousing function. It tracks inventories at up to 63 warehouses for up to 200 distinct, numbered items. It provides individual invoices for each sale and periodic reports for each account showing all purchases and payments. Daily reports are generated for stock on hand and resupply requirements by warehouse.

Each warehouse station consists of a Commodore 64 computer, a 1541 disk drive, and a 1525 printer. Its information base is supplied each day on a single floppy disk delivered from the central system. A transaction is begun by entering the account name (or "walk-in"). Each item's name and quantity are entered into a dummy invoice on the screen. The account's discount and tax rates and the item's piece and total costs are displayed. Accepting that item readies the system for the next. When all are input, entering an item name "total" prints the invoice for that purchase and logs the sale. Items may be identified by name or number. Returned items are treated as transactions in the same way as purchases, except that inventory is not incremented by the return. Items are added to inventory as delivery transactions. A second function of the warehouse station is to check both local and distant inventories for availability of an item; local inventory is maintained current including that day's sales and deliveries; remote inventories are supplied by the central system and are current to the previous day. The system is expandable for telephone communication of daily activity, although the baseline system requires carrying floppy disks between warehouses and the central station.

The central system requires a Commodore 8032 computer, 4040 disk drive, and both 8300 and 4022 printers. It integrates the daily logs from the warehouses and reports inventory status and resupply requirements. It accepts comprehensive inventory inputs, payments on account, and corrections to prior transactions and payments. The central system also maintains all system information, including account names, addresses, discount rates, tax rates, and credit limits. Periodically, the central system generates bills for all accounts or for a named account (user option) and a report of activity by warehouse. The central system may also serve as a warehouse station, but only when central operations are not required.

Some parameters are necessarily limited in the ICS. They include maxima of 63 warehouses, 200 enumerated items, and five price groups per item.

6.5 Design

Let's suppose that the customer has accepted the Inventory Control System as described, and has agreed to your terms for development (including payment schedule). Since most of the eventual system cost will be hardware, you should establish a link with a local dealer; a dealer's cooperation will be easy to obtain and may simplify some of your problems—including getting information and hardware. Your next task is to lay out the files and programs of the system in detail, to schedule your work so you'll know whether you're in trouble or not, and, finally, to start coding.

Transactions are collected in the daily log, which carries the coded date in its name. Each transaction has a type (cash sale, credit sale, return, delivery), account, part number, quantity, line-item cost/credit. Someday the system may expand to include salesman commission, so leave space for an ID field.

Inventory on disk is a relative file with two records per warehouse. (200 items at 2 bytes each won't fit into a 255-character maximum record.) The item names and numbers are in a sequential file read in at the start of the day; one each string and integer arrays required, at about 25 characters/item for 5K storage. We'll keep the local inventory in RAM as well, so we can update it as the day's transactions proceed. Brute force storage (200 items × 63 warehouses × 7 bytes) takes about half the disk. We should be conservative and pack inventory into two bytes per entry (use the 0-200 counter twice for a maximum of about 40,000 items). We can afford to keep that file on disk and to spend time to unpack it since it is accessed only rarely—when the customer needs more widgets than this warehouse has in stock.

In contrast, the list of accounts is searched for every purchase. Figuring the average account name as 15 characters, we could handle 200 of them in 3K. *Note*: Add the 200-account limit to the system constraints. We'll pack the number in the usual way. If the number of accounts goes over 200, we'd need two characters for the id—no problem. If it goes over 500, we'd need more than 7.5K of RAM—that'll be a problem. If they can live with 200, that's the design. The other account data can be relative and fetched when needed (once per invoice).

Program architecture for the warehouse is almost trivial. After initialization, enter the account (or a dummy for a delivery). Create a display with account information, and start the invoice page. Select transaction type, accept input data, print the line and log the transaction. Then return for the next transaction. If the transaction type is "total," present final display for confirmation, then wrap up the invoice on the printer. (Check invoice formatting; special paging may be needed to get the form out of the printer without too much wastage. How many transactions on a typical order?) A dummy account can be used for "inventory"—to trigger an inventory check. Its logic is to report first how many are in stock here. If more are needed, the warehouses with at least the required (input) quantity are on hand.

One module handles the inventory check. Otherwise, all transactions are equivalent, with variations depending on whether inventory is decremented, incremented, or left unchanged. (The idea of not putting returns back in inventory assumes that they will be examined for damage or return to shipper. Check with customer on a "return to inventory" option after each return. If so, implement with another transaction type, specified at its completion.) Select cash or credit at the end of the transaction (during total on a sale); that means holding all disk writes until acceptance, which we would want to do anyway. It also means a limit on the number of transactions per invoice—get the data from the customer. (Answer comes back: maximum of ten lines per invoice, automatic heading of second page with appropriate labelling of successor pages, but each page is subtotaled and accepted as a unit. Otherwise, there's too much risk of redoing ten minutes' work for one typo.)

The process of developing the design is repeated for the more substantial central programs. As the concepts emerge, questions arise that require answers from your customer. Your relationship is now close enough that occasional calls for information not only will be acceptable, but they'll be welcome to indicate your progress. As soon as the design is complete and the questions have been asked and answered, you're ready to revise the customer description and to draft the User Manual.

6.6 Using the warehouse program

Each day's operation begins by turning on power to the computer, the disk drive, and the printer. Insert the new disk into the drive (label side up), and enter:

```
LOAD"*",8 <C/R>
```

The red light will go on and the warehouse program will be loaded from the disk. Then type:

```
RUN <C/R>
```

As the program starts to run, it reads information from the disk into the computer. It then tells you the date for which that disk was designed, and asks you to "Confirm" it; if it's right, just answer "y" ⟨C/R⟩. If you are using another day's disk—say because today's is late—answer "n," then type in the date as month/day/year.

The screen now clears and asks you to enter the "Account." You may type in the account name or number, "inventory," or "delivery." If you enter only the first part of the Account name (like "Adams" for "Adams and Smith Fine Furniture"), the program will show you all the accounts beginning that way ("Adams") with their account numbers, and ask you to type in the number of the one you want. A number 0 tells the computer that none was the one you wanted, and you'll reenter the account. Once the account is specified, the screen will show you the full name, discount and tax rates, and credit data. The printer will start to run, putting out the top part of the invoice. Then you type in what item is being purchased or returned and how many pieces. The items may be specified by part number, full name, or part of the name, just like the Account. The screen will show you the price per item and the total price for that quantity, and ask you to "Accept" the entry. If you enter "y," it's ready for the next. Answer "n," and that line item is cancelled. When you have entered all items, call the next one "total," and the computer will figure the total with tax and discount and show it to you for you to "Approve." If you do, you tell whether the sale or return is for "Cash" (otherwise, it is handled by crediting or debiting the account). The invoice is completed and filed automatically. If the sale was not accepted, the printer voids and ejects the page before asking for the next "Account."

Notice that you have to treat an exchange as two separate operations—one for return, the other for sale—and you have to enter the Account twice. If the order is for more than ten items, after every ten the computer will ask you to confirm the work so far and will automatically subtotal and complete that invoice page. When you finally run a total, the subtotal for the last page will be computed and printed as well as the grand total. Remember that each page is accepted separately, so you have no way to correct an earlier page of a large order except by creating a return invoice for items your customer decides not to take.

For walk-in customers, the Account is "walk-in," number 0. Of course, there is no walk-in account as such, so it cannot be credited or debited, and all transactions are automatically for cash.

If you call the Account "inventory," you will be asked "What part?" Enter its number or its full or partial name. The screen will tell you how many are in stock. If you need more than you have on hand, answer the question: "Check other warehouses?" with "y." Then enter the number of pieces your customer needs. The program will list the numbers of the warehouses that have enough to cover the order, so you can either send the customer to one of them or arrange to pick them up for him.

When you're through for the day, remove the disk from the drive and pack it for shipment to the central office. Take the spare disk from the file and insert it in the drive. Then enter "quit" for the Account, and the spare disk will be updated with your current inventory. Put the spare away, shut off the power, and call it a night. The next day, a new disk should be ready and you repeat the process. If the new disk has not arrived, use the spare instead. You will have to tell it today's date, and the inventories for the other warehouses won't be up to date, but it will do everything you need and central can sort out the problems.

6.7 Design review meeting

With the draft User Manual in hand, you're ready to hold a design review with your customer and her user(s). Deliver the manual a day or two before the meeting, and try to have someone read it through. While they're reading, you're writing—the test plan. For the three programs in the ICS, you may want one, two, or three reviews. After all, the user of the warehouse program will be a salesperson, not a computer operator; the clerk running the daily program at the central station will have more familiarity with the computer; and the bookkeeper who runs the monthly program has specialized needs. The customer will be at all three meetings, since she has to approve your going ahead with coding. The user of each program should be at the appropriate session. Realistically, you will have to work to make sure that the design reviews aren't just rubber stamps. Usually, no one has really read your material, no one is ready to ask meaningful questions, and the review turns into a formal process for putting the customer on the hook—committing her to your plans.

The first step in the review is to run through the User Manual to make sure that everyone understands it. Keep it simple and direct, but identify everything that the program will do and everything that might be relevant that it doesn't. Make sure that they understand that disks have to be carried back and forth each day for good results, that only 200 accounts can be active, and any other limitations. Then pull out the test plan and show them what you'll be testing and how. Ask for input, comments, or questions. Your objective is to leave the meeting with general approval, specific changes to be made, and the customer feeling that it's *her* system already.

Try to avoid broad, open action items. In the meeting, tell them what you will change to implement anything that you accept. If they ask you for a major redesign, point to the Customer Description that they already accepted. Most changes will modify that agreement, and give you the right to change your cost and schedule. Did they reconsider the 200-item limit and decide that they want at least 1000? Your whole design may be broken by that "simple" change. If the customer insists on it, your options are to submit revised documentation, schedules, and fee or to terminate the contract and request payment for services already rendered—in accordance with your agreement. Be firm, explain the problem, but don't give in. To handle 1,000 items will take 20K of RAM; the 64 will be crammed, garbage collection will be prohibitive, and the system won't run adequately. The disk files will become massive, and the fragile 1541 may not stand the burden. For a thousand entries, you may be forced to a completely different system—hardware and software. Help them to understand what that means, and why the change may be fatal.

Both you and your customer want the system to work, so bend where you can and give them what you can afford. Try to postpone even minor extensions until the system is running. Diverting your attention from the fundamental problem threatens to delay delivery and to degrade the product. Make notes, but no commitments, on each change that they request. Promise nothing that you didn't plan to do when you designed the system. Be as friendly as possible, but don't give in.

Hand out copies of the test plan during the meeting, and walk through it with your customer. Point out that you're testing everything the User Manual says, both positive and negative. Invite them to suggest additional testing, and ask the user to participate. If he's willing, let him join you for the testing itself. Again, the customer will have adopted your program as her own even before delivery.

Finally, pin down some dates at the design review. Schedule delivery of the documentation with the changes you have agreed to. Get a commitment for their review, confirmation, and signoff by a fixed date. Make sure that they understand that you cannot proceed without those signatures.

When the revised User Manual and Test Plan are ready, deliver them with a cover letter that includes a list of the changes you made due to the review, any changes in cost and schedule that they have caused, and the date on which you require signoff. Resist the temptation to put in a negative option (" . . . will be regarded as accepted unless . . . "). It's much weaker than a signature, and the User Manual is your specification (hence your contract) while the Test Plan embodies the acceptance criteria (the way you know you're done).

6.8 Coding

Now that we've come to the enjoyable part of the process, there isn't much to say. Coding is a highly personal process; each programmer has his or her own approach. Some prefer to write a routine of a dozen lines or so and check it out thoroughly before they proceed. Others insist on getting the whole thing down before the first RUN. If you prefer, work the modules one at a time; an alternative is to label all of them, then write one and all that GOTO it, then all that it can reach, and so on. All that matters is that you use some system that you find comfortable and efficient.

Common sense suggests that you start with a nucleus of initialization and utility routines from earlier programs. If there are utilities you haven't tried before (say, RECORD in BASIC 2.0), code them separately and run some tests. Since part of the system will run on a 64, figure on BASIC 2.0 for all of the warehouse program; while you're at it, stick to 2.0 throughout for consistency and ease of maintenance. You'll probably develop the program on another host (4032/8032) to take advantage of your programming tools and the IEEE buss, and you could fall into BASIC 4.0 without thinking. Try to drill BASIC 2.0 into your head so you don't slip accidentally.

An experienced programmer doing a variation on an established theme may spend half the time coding, half in debugging and testing. The less experience you have with that type of problem, the longer testing will take. No customer will be upset if you deliver ahead of schedule. If coding will take four weeks, schedule six for checkout and test. Maybe you'll finish in four, or even three—great! But schedule for four and run into one extra bug, and you'll lose your schedule or even your contract.

Remember your objective: deliver a quality product. Don't compromise in testing to meet the schedule, and don't deliver code you haven't fully checked out. If the user joins you in testing, you'll save no time; you will have an in-house supporter to attest that you fulfilled your contract. If you have to test alone, invite your customer to witness the final run-through of the plan. At least, ask her to select some parts of the test to witness. Throughout testing, and particularly during the final pass, document every result and its results compared with those you predicted in the test cases. Once you have your customer's signoff on the test cases, the system is finished. Unfortunately, your job is not.

While coding, you will find improvements over your original design. Many will simply be better ways to do the original job. Some will modify the procedures in the User Manual—or even the capabilities in the Customer Description. Within the latitude provided by the approved documentation, the design is entirely under your control. Change it to save memory, disk space, running time, or difficulty in development. If your improvements will change the documentation, you cannot proceed without customer approval. Most changes will give a better product, and she will buy off quickly. Some may be necessary to meet schedule (budget is your problem, not hers) even though they will cost performance; those will be harder to sell. As soon as you see a design change that affects documentation, talk it over with the customer to get verbal approval. Before you start to test, get her signature on the revised documentation—including the Test Plan. You must have final documentation in hand before you start acceptance tests and demonstrations, for your protection and hers.

6.9 Maintenance

Software cannot wear out or break; if it was correct when it was written, it cannot go wrong. In hardware, maintenance means cleaning, inspection, and minor repair. In software, it means repair and expansion. Once the program is delivered, you can expect calls of three varieties:

"It doesn't work. Come fix it."
"It doesn't work the way we want."
"Can't we make it do this, too?"

They correspond to three different kinds of changes, and require different responses.

If the program is not performing as specified, it needs repair. Both morally and contractually, that's your responsibility for a fixed period after delivery. However comprehensive your testing was, a few weeks of the real world will tax the software in ways you didn't cover. Even if your program is right, there will be hardware glitches—noise on the power line, faulty disks, and something never seen before just to complicate existence. Since your product for the Widget Company was a system, you're stuck with hardware problems as well as software. If you had put software into existing hardware, you would be less securely on the hook.

When you get a trouble report, the first (and toughest) problem is to find out what really happened. Your experience and the user's garbled report of the phenomenon are all you can count on. You *must* get the system back into operation, even if it requires repeating operations. Frequently, you will have to watch the user experience the problem. First, tell him how to rebuild the system (it should be in the User Manual, but tell him where to look) and see if the problem repeats. If it doesn't, and if it does not recur too often, blame it on power lines and track it no further. If it repeats, try it yourself or watch the user do it a third time. Diagnose and solve every repeated problem. If the program and procedures were correct and the problem stemmed from misuse of the system, the fault is the user's and you are entitled to be paid for your consultation. If the fault is in the program or the User Manual, you are responsible for its correction as part of the original contract.

Limit your liability in the original agreement so that you are not on the hook for consequent costs; loss of business and fouled records can be very expensive to your customer. Whatever the contract says, you may have problems if you did not take all reasonable measures to protect against faults and to limit their effect when they occur. You were employed as an expert, and if you did not use expert methods you are responsible for failures. Before you sign up for software on which a company will depend, check with an attorney.

Most requests for maintenance stem from things the user wants to do that the system wasn't designed to handle or that he doesn't know how to accomplish. If it can do what he wants, just tell him how. If it can't, explain that he is going beyond its design limits. Widget's system will handle 200 items, and they just got their 201st. Now what? Are all 201 really active now? If not, remind them that they can replace outdated ones with current products. But if they really need 201, there's a problem. You used a coding scheme that won't expand, and made it clear from the beginning that the limit was solid. The problem cannot be "fixed" by changing a few lines or modifying procedures. It has just escalated beyond what the user can handle; call in the customer.

It's surprisingly difficult for a customer to recognize that a system that works well for 200 items just won't handle 201. You explain that both the disk files and the program have to change, and the reaction is, approximately, "Change 'em!" She should recognize that there is no instant fix, so the problem will last for some weeks until you can make some minimum modifications. She has to pay for a week or so of your time to patch in emergency measures, then may have to go to a whole new system to handle the growth. Explain the process that you have to go through so she understands the fact that it's complex and time consuming.

What will you do? To begin, you need a two-byte item ID instead of one. The extra code is easy, but you need a program to transcribe and expand each file, inserting a "0" first digit. You can't afford to require a new inventory and manual reentry just because you've changed the file structure. Finally, you impose a new limit and implement it by changing array dimensions. The new numbering system would handle up to 40,000 items—but the computer can't. With luck, the running programs will have room for expansion to, say, 250 items. That will only handle growth for a few months. For the Widget Company to stay in business, more extensive changes are needed—in a hurry. Get customer agreement, rework your schedule, and patch up the code. However great their panic, retest the system before you deliver it. They will have worked around the problem while you patched, so they can live with it for another day. They cannot live with a patch that breaks down a week later.

When you deliver the modified system, stress the need for redesign to handle the growing load. The customer must understand that you cannot stretch the hardware any further, and that there's barely time to adapt before the system saturates again. If she isn't ready to face the problem, tell her that she hasn't much time, then go home and write a letter. Formally identify the facts: the patch is temporary, but functional; the system will not handle further growth without substantial redesign; you cannot be responsible for the effects of delaying that redesign. Make sure that you are off both the legal and the moral hooks. Put the problem on paper and be certain that your customer receives it.

If the customer is ready to face reality, take a contract to find practical, economical solutions. Suppose that the 64's have space, but the 8032 just can't handle more items. Maybe there's a new machine with full compatibility and more RAM. Perhaps the critical 8032 code could keep the files on a hard disk and access them there, saving RAM. More likely, the 64's have filled to the point where garbage collection becomes prohibitive. Switching computers is expensive—it means new disk drives, too, and there are at least 30 stations to resupply. A week should be time enough for you to construct a maximum system for the hardware you know, to estimate cost and schedule for several alternatives, and to prepare the sad message for your customer. With customer approval, contact a supplier of minicomputer systems. If you exceed the capacity of a micro, maybe the Widget Company has to move up in capacity and price.

When you started the process, you had to guard against contagious panic. That rule applies here as well. There is no greater trap than promising more than the system can deliver. There are few worse things to do to your customer than make her believe that an unsolvable problem is under control. Even when a problem has outgrown your programming, a customer will still need your expert consultation. She'll still be a customer.

6.10 Bigger and better

Once the system is up and running, you are in an ideal position to do more for your customer and yourself. The system itself may not have room for growth, but the computer can serve other purposes than those for which it was bought. The key word is "noninterference." What can be done with the system and the files that will not interfere with ICS but will make the Widget Company more profitable?

The first thing to do is to ensure that there is hardware available for the extra work. Those thirty 64's are well used all day, but they should be free on second shift. The central system is in use at all times, but the backup is idle except when replacing a failed one. If the backup and its 4022 printer are used for the central warehousing job, then a full system with 8032, 4040, and 8300 is available for most prime shifts. What can we propose to Widget for that hardware?

First, there are standard software packages—spreadsheets, word processors, general ledgers, and many more. They may be taken off the dealer's shelf and run as is. There is little for you to contribute except to advise Widget about what's out there, how to select, and how to use the products they choose.

But there are greater benefits for both sides if you involve yourself. For example, you may run WordPro as it comes for excellent word processing. But you'll find it hard to tie it in to the ICS files. It's quite a job to generate a cover letter for the monthly statement that uses all the good stuff the system knows. Suppose you start with the Word Machine. It's inexpensive and written in BASIC. Rip it apart and interface it to the ICS files. A little imagination will let you adapt the program so it cites balance past due, late-payment charges, and other items as they are relevant to that account.

The same sort of thing applies to a general ledger or accounting package. Half of the accounting is already implied in the ICS files—items sold and payments received. Look for a package that you can link into those files so that bookkeeping will require entering information only once. Be willing to sacrifice elaborate features with low payoff to Widget Company for easy interfacing. When you have finished ICS, you will be in a unique position. You will know more about how Widget Company could use computers and more about computers and software that apply to Widget than anyone else connected with the company. Your consultation is valuable to management, and they'll probably be well aware of it—and willing to pay for you to use it.

Don't confuse yourself by working the second problem first. Your primary job is ICS. Get it working without concern about WordPro or General Ledger. Only after you have a satisfied customer should you spend time or thought on what else can be done. After your first few projects, you will automatically organize your systems for easy interfacing to your favorite purchased products. But don't be tempted by the glitter in the distance to leave the development vein you're paid to mine. The time to look for building onto the system is after both you and your customer are sure that it's working.

Programs Supplied on the Diskette

Seven programs are provided on the e~~nclosed di~~sk. All are referenced or contained in the book *Advanced BASIC Programming for the Commodore 64 and Other Commodore Computers*. All programs are self-documented, so user instructions are not required.

SUPERLIST is the principal program developed in Chapter 5 to provide cross-referenced listings.

DIABLO SUPERLIST is the variant of SUPERLIST for letter-quality publication copy. It is described in 5.7 SUPERLISTing and was used for all listings in the book.

SYS CHECKSUM is developed in 5.8 to show the use of embedded machine-language code in a BASIC program.

CHECKSUM is a BASIC-only checksum program intermediate between SUPERLIST and SYS CHECKSUM. It is referenced in 5.8 and included on the disk to clarify the relationship between its parent and its descendant.

PROGRAM COMPARE reports line by line on the differences between two BASIC programs. It assists the advanced programmer in tracking down differences in code which have been flagged by different checksums.

TURTLEWALK is developed in 5.9 as an example of an educational program for microcomputers.

DOMINOES is discussed in 5.10 as a less-advanced program. It is provided on the disk as a working program for the student to improve, applying the principles developed in the book to create a better product.

Index

Absolute address, 27
Account(s)
  entering of, 107–108
  list of, 106
Accounts receivable, 104–105
Addition, 34
Address(es), 26, 32, 41, 72
  absolute, 27
  return, 27
  secondary, 49
  of variable, 29, 30
Addresser, for SYS CHECKSUM program, 82
Advanced BASIC programming, 1–2
  prerequisites and purposes, 2–4
Advanced programmer, writing a program for, 9
Advanced software, 1
  interoperability, 5–7
  writing of, 97
ADVENTURE, 2
Alternative designs, 13
AND, 41, 44
Anomalies, 25
APPEND command, 58
  use, 11
Apple machines, 1, 2
Application program, bytes, 25
Arithmetic, 33
Array(s), 31–32
  defining of, 32
  dimensioning of, 6, 32–33
  multidimensioned, 13
  storage required, 32
  structure, 11, 12
Array name, 32
Array "variable," 75
ASC function, 28, 33, 34, 37
ASCII, 33, 43, 53, 60, 75
  changes in, 50–51
  characters, 28
  conversion to, 41
  in SYS CHECKSUM program, 83
  values, 34, 36, 39, 72
ATN command, 28
Automatic line feed, 49

Backup, 11, 114
BASIC, 1, 85
  information on, sources, 2
  principles, 4–5
  start of, 27
  use, 5
BASIC AID program, 41
BASIC commands, *see* Commands
BASIC programming, advanced, *see* Advanced BASIC programming
BASIC 2.0, 111
  garbage collection, 25
Beginner programmer, writing a program for, 9
Bells, 37
Binary, 33
Bit(s), 33
  assigning of, 33
"Bones," dominoes and, 90
Bookkeeping, 115
Brackets, 77
Branching, 37
Brute force storage, 106
Buffer
  cassette, 40
  keyboard, emptying of, 36
Bug, *see* Error
Buss system, 50
Byte(s), 25, 33–34
  high-order, 27
  of a line, in SUPERLIST program, 72
  losing of, 14
  meanings, 34
  saving of, 28, 29

cA command, 28
Carriage return, 52, 56
  automatic, 55
Cartridges, 52–53

Cassette, 40
  dual, 47
Cassette buffer, 40
CATALOG, 28
Catalogue, software for, 65
Catalogue system, 97, 98
C-BASIC, 2
CBM, 6, 47
Central station, 104
Central system, 114
  function, 103–104
  for inventory control, 105
Chaining of program, 43
Character(s)
  ASCII value of, 36
  on dot-matrix printer, 52
  nonprintable, 69
  reverse, 6
  routine for getting, 36–37
Checks, crediting of, 104
Checksum, 68, 71, 72, 74
  see also SYS CHECKSUM program
chr$, 49, 52, 59
  meaning of, 34, 48
CHR$ command, 34
  use, 38
Clarity, 14, 16
Clocks, 50
CLOSE function, 48, 49, 58
CLOSEn, 17
CLR character, 14, 48, 52
CLR function, 42, 43
CMD, use, 47
COBOL, 1, 4, 68
  interpretive versions, 4
  use, 5
Code(s)
  blocking of, 18
  clarity of, 16
  lines of, 12
    sequence, 26
  utility, 13
Code number, 9
  printing of, 67
Coding, 39
  process, 110–111
  tricks, DEF and ON, 43–45
COLLECT command, 58, 59
Colon(s), 29, 38
Color, 37
Comma, 38, 73
Command(s)
  beginning of, 26
  disk, 40, 57–59
  processing of, 74
  types and use, 28–29
Commodore 64, 1, 50
Commodore computers, dimensions of variations, 5–6
Commodore's Word Machine, *see* Word Machine
Communication, establishing of, 47
Compilation time, 4
Compiled languages, structured programming for, 17
Compiler(s), 4
Computer games, 2
  DOMINOES, 90–96
CONCAT, 58, 59
Confusion, 28
CONT, 42
Context, of information, 35
  changing of, 34
Contract, 112
Conventions, international, varying from, 49–51
COPY function, 59
Counter, use, 73
CP/M, 2
CPU
  items communicating with, *see* Devices
  time, 50
Cursor, dummy, 36
Cursor-control character, typing of, 6
Cursor-down, 37, 71
Customer, 3, 4
  first call from, 98–99
  first meeting with, 100–101
  needs, 100–101
  program documentation for, 19

Daily activity, 103
Daily program, 104
Data
  global, 14
  loss of, 16
  packing of, 34
Data base, 25, 64
"Data base manager," 64
Data definitions, 21
Data-entry professional, writing for, 9
Data files, 63
  designing of, 63
Data flag (df), 74
DATA line, 14

DATA statement, 14, 73
  placing of, 14
  search for, 14
  in SUPERLIST, 69
Data structures, structuring, 13, 18
Datasette, device number, 47
Debugging, 42, 111
Decimal, 26
DEF, use, 43–45
Default(s), 47
Default device, changing of, 47
Default values, 47
DEL function, 37, 70
Delimiters, elimination of, 38
Design
  improvement of, 111
  for warehouse program, 106–107
Design review, 101
  meeting, 109–110
Device(s), 47–49
  disk commands, 57–59
  disk and drives, 53–55
  diskettes, 55–57
  printers, 51–53
  sorts and searches, 63–65
  tokens, 59–61
  un-conventions, 49–51
  using files, 61–63
Device numbers, 47
df, 74
dI, 28
Diablo printer, 51
Dialects, 2
DIM command, 28
DIM(X) command, 32
diR command, 28
Directory
  disk, 55
  modifying of, 58
  problems, 58
DIRECTORY command, 28
  token for, 59–60
Disk(s), 53, 65
  device number, 47
  faults, protection against, 11
  faulty, 53–54
  floppy, 54–55, 61
  handling of, 53
  hard, 54, 55, 61
  inventory on, 106
  reinitializing of, 58
  status of, 58
  storage on, 11
  use, 3
Disk access, slowing of, 16
Disk channel, 55
Disk commands, 40
  sending of, 57–59
Disk directory, 28, 55
Disk drive(s), 53
  types and features, 54
Disk error trap, 15
Disk file, 31
  closing of, 74
  types and features, 55–57
"Disk id mismatch," 58
Disk operating system (DOS), 55, 56
  loss, 58
Disk-resident data file, 57
Display
  for references, 74
  on screen, 72
Display features, 37
Documentation
  delivery, 101
  program, 19–21
DOMINOES program, 2, 90–96
DOPEN, 76
DOS, *see* Disk operating system
Dot-matrix printer, 16, 77
  features, 51–52
  operation to, 48
DTL, 4
Dual-cassette capacity machine, 47
Dual disk drives, 54
Dual-drive system, 58
Dual-tape machines, 40
Dummy cursor, 36
Dummy file, 56

ELSE structure, 45
END command, use, 41, 42
Endings, of program, 41–42
EOR instruction, 82
Error(s), 59
  analysis, 25
  file, 48
  handling of, 15, 22
  minimizing of, 10–11
  "next without for," 38
  "out of memory," 27, 38
  syntax, 26, 73
ESCape sequence, 52
Exit, single, 17
Expansion of system, 114–115
Exponentiation, 34

"False," 43, 44
FCL, *see* Functional Capabilities List
Feature(s), adding of, 17
Field(s)
  adding of, 98
  preplanning of, 13
File(s)
  closing of, 12, 17, 47
  disk, 55–57
  opening of, 49
  tokenized, assessing of, 60–61
  updating of, 64–65
  using, 61–63
File error, 48
File manager, building of, 65
File numbers, assigning devices to, 49
File operations, 48
File structures, 11–12
  choosing of, 65
Flag(s), 33, 34
  checking of, 16
  data, 74
  initializing of, 72
  line, 72, 73, 74
  quotes, 72–73
  setting and testing of, 17, 75
  variable, 73, 74
Floating-point number
  conversion to string, 35
  storing of, 34
Floppy disks, 54–55, 61
Flowchart, 12, 18, 21
  high-level, 13
  replacement of, 18
FN command, 75
FNP function, 44
FNU function, 44
FNX function, 69, 75
FOR definition, function, 75
FOR . . . NEXT loop, 26, 32, 72
  use, 38–40
Formatting, 49
  within printers, 52
FORTH, 4, 5, 68, 85
FORTRAN, 1, 4, 5, 68, 85, 86
  costs, 1
  interpretive versions of, 4
Fourier transformation, 5
FREe memory, 6
  setting working variable to, 69
Function(s), user-defined, 44
Functional Capabilities List (FCL), 22

"Garbage," 31
"Garbage collection," 25, 31, 69
General ledger, 114, 115
GET command, 15, 47, 70
  use, 35–36
GET function, 38
GET# command, 47, 55
Global data, 14
  definitions, 11, 14
GO, 75
GO TO command, 75, 76
GOSUB command, 12, 14, 15, 18, 26, 28, 67, 72, 74
  operation, 27
GOSUB20:RETURN, 37
GOTO command, 12, 14, 15, 16, 26, 27, 28, 49, 67, 72, 74, 110
  avoiding of, 18
  bytes used, 28
  "computed," 18
  use, 17, 39, 42
GPIB, 50
Graphics, 48
  interoperability, 6
  printout, 71
Graphics character, 76
Graphics mode
  interoperability of, 6–7
  selection, 49

Hard disks
  features, 54–55
  tokenizing and, 61
Hardware
  costs, 104
  maintenance, 111, 112
HEADER command, 58, 59, 76
Hexadecimal, 26
  counting in, 43
High-order byte, 27
High-speed utilities, 15
HOME character, 6, 48, 52
HPIB, 50

IEEE buss, 111
  disk drives for, 54
  printers, 51
IEEE printer, 52
IEEE system, 104
IEEE 488 busses, 50
IF command, 29
  use, 39
IF test, 44
IF . . . THEN, 44, 45
IF . . . THENGOSUB, 45
Increment (STEP), defaults, size of, 38

Information
  attaching of, 58
  changing contexts for, 34
  codes collecting, 13
  context of, 35
  gathering, for pursuing of project, 101
  holding of, 34
  losing of, 10
  minimum set, 11
  order of entering, 10
  set up, 25
  storage, in array, 32
  updating of, 16
Initialization, 15, 110
  bytes for, 25
  SUPERLIST, 68–70
Input, keyboard, 35–38
INPUT command, 47, 48, 60
  changing operation of, 48
  use, 35
INPUT variable, value of, 48
INPUT# command, 43, 47, 56
Instructions, interpreting of, 4
Integer array, 73–74
Integer variable, 30
Interest rate, 104
Interface, interfacing, 53
  high-speed, 50
  machine language, 40–41
  of system, 103
  to outside world, 47
Interoperability
  of advanced software, 5–7
  maximizing of, 75
Interoperation, 16
Interpreter, 26, 27
Interpretive language, advantage of, 4–5
Interview, customer, 100
Inventory(ies), 103
  on disk, 106
  review of, 102
Inventory control (customer description), 104–105, 106, 107, 109
Inventory Control System (ICS), 97, 104–105, 114, 115

Jargon, 17

Keyboard, 48
  device number, 47
  input, 35–38
Keystroke problem, solution to, 10–11
Keywords, 28

Language(s), 1
  interpretive, 4–5
  modern, 4
  professional, 5
  properties, 5
Ledger(s), 114, 115
Letter-quality printer, 53
lf, 73
Line(s)
  discarding of, 67
  formatting of, 75
  processing of, in SUPERLIST program, 72–74
Line feed, automatic, 49
Line flag (lf), 72, 73, 74
Line number, 28
  bytes used, 29
  cross referencing of, 67–68
  features, 26–27
  handling of, in SUPERLIST program, 72
  massaging of, 74
  references, printing of, 75
Line references, cross referencing variables by, 68
LISP, 4
LIST command, 28
  in SUPERLISTing, 76
Listing, features of, 67–68
LISTing, 82
  device for, 47
Literals, 48
Literals mode, 7
  selection, 49
LOAD command, 42
  function, 43
  use, 43
Log(s), saving of, 103
Logical expression, evaluation of, 43
Loop(s)
  initiation, 73
  see also FOR . . . NEXT loop
Loop index, 37
Low-speed utilities, 15
Lowercase letters, ASCII conversion for, 50, 51

Machine, *see* Computer
Machine code, for SYS CHECKSUM program, 82

Machine-language routine, location of, 40–41
Macros, 85
Mailing-list program, 32
Mainframe(s), timing requirements, 17
Maintenance documentation, 20–21
Maintenance Manual, 20
Maintenance of system, 111–114
"Map" of memory, 31
Margins, wide, obtaining of, 77
Material, review of, 22
Mathematical computation, 5
Mathematical problem, 32
M-BASIC, 2
Mechanics of a program, 25–26
  arrays, 31–33
  bits, bytes, characters, and numbers, 33–35
  coding tricks, DEF and ON, 43–45
  commands, 28–29
  FOR . . . NEXT loops, 38–39
  keyboard input, 35–38
  line numbers, 26–27
  machine-language interface, 40–41
  odds and endings, 41–43
  variables, 29f, 29–31
Meeting(s)
  with customer, 100–101
  design review, 109–110
Memory, 3, 6
  balance with time, 26
  losing of, 10, 14, 29, 31, 38
  saving of, 25, 26, 34, 45
  size, 5
Memory map, 40
Menu, 9
  starting of, 16
Merge, 58
Microcomputer(s), 19
  timing requirements, 17
Microprocessors, 50
Microsoft, BASIC from, 2
MID$ function, 33
Modem, 103
Modern languages, 4
Modifying the program, 15
Modularization, 25
  function of, 12
  process of, 12
Modulator, 3
Module(s), 18
  lay out of, 14
  location of, 16
Monitor, monochrome, 3
Monochrome computer, 5, 6
Monochrome monitor, 3
Monthly program, 104
Multidimensioned array, 13
Multiplication, 34
Murphy's law, computer user and, 10

Name Machine, 65
Needs
  customer, 100–101
  programs based on, 25
Nested routine, 17
NEW command, 42, 59
  use, 41
NEXT, *see* FOR . . . NEXT loop
"Next without for" error, 38
Nonprintable characters, translation, 69
Novice programmer, writing programs for, 9, 10
Null string, 48, 71
Number(s)
  floating-point, 34, 35
  operation on, 35
Numerals, shifted, 76
Numeric function, unary, 43
Numeric values, 43
Numeric variable, 30
  storage, 30

ON, 26
  use, 43–45
ON . . . GOSUB, 18, 44, 74
  use, 45
ON . . . GOTO, 18, 44, 73, 74
  use, 45
onjgoto, 37
OPEN command, 11
OPEN1,8,0, "fname," 55
Options, selection of, 9
OR, 41, 44, 69
Organization of program, 14–17
"Out of memory" error, 27, 38
Outside world, interfacing to, 47

p$string, 71, 75
"Packing" of data, 34
Paging, on dot-matrix printer, 52
Paging string, 48
Parenthesis, 28, 73
PASCAL, 1, 4
Pattern-matching capabilities, 71
PEEK function, 34, 40

Peripherals, 48
  development of, 50
PET, 47
Petspeed, 4
pi, 77
Pilot, 85
Planning of program, 11–14
Pointer, 26, 71
Poke, 6
POKE command, 31, 40, 41, 48
  in SYS CHECKSUM program, 83
Popping, 39
Power lines, faults, protection against, 11
Prefix string (s$), 71
Prerequisites, for advanced BASIC programming, 2–4
Price structure, 102–103
Principles of BASIC, 4–5
PRINT command, 28, 47, 48
PRINT# command, 11, 28, 47, 48, 55, 56
Print string (p$), 71
  in SUPERLIST, 75
PRINT TIME command, 30
Printer, 3, 15, 49, 114
  capabilities, 75
  checking of, 10
  device number, 47
  dot-matrix, *see* Dot-matrix printer
  problems with, 77
  speed of, 16
  types and features, 51–53
PRINTing, device for, 47
Printing, of references, 74–75
"Professional" languages, 5
Professional software, 3
Program
  blocks, 11, 12
  characterizing of, 68
  concept, teaching of, 85
  documentation, 19–21
  mechanics of, *see* Mechanics of a program
  organization, 14–17
  planning of, 11–14
  printed listing of, *see* SUPERLIST
  testing of, 21–23
PROGRAM COMPARE, 77
Program file, 55
  features, 55
Program name, entering of, in SUPERLIST, 71
Programmer, 19
Programmer documentation, 20
Programming, structured, 17–19
Project, pursuing of, 97–98
  coding, 110–111
  design, 105–107
  design review meeting, 109–110
  expansion, 114–115
  first call, 98–99
  first meeting, 100–101
  inventory control (customer description), 104–105
  maintenance, 111–114
  structuring the system, 102–104
  using warehouse program, 107–108
Protocol(s), 47, 48, 49
Punctuation, 76
Purposes of advanced BASIC programming, 2–4

q$, 73
Question mark ("?"), use, 18, 28
Quote array (q$), 73
Quotes, 6, 28, 76
  information in, 73
Quotes flag, in processing a line, 72–73
Quotes mode, 6

RAM, 25, 31, 64, 106
  saving of, 114
  top of, 40
  wasting of, 60
READ command, 14
Reader, classes of, 19
Reading of file, 58
Record(s)
  addition and deletion of, 63
  keeping of, 13
Redimensioning, of arrays, 32
Redundancy, 11
Reference(s), printing of, 74–75
Reference processing, ending of, 74
Relative file, 57, 65, 98
  features, 56–57
  for inventory control, 106
  use, 57
REM, 14, 82
  insertion of, 40–41
  use, 14–15, 20
REMark(s), 29, 72, 73
REMark line, 15
RENAME, 58
"Reserved" word, 14
RESET, 41

Resolution, 3
RESTORE, 14
Retrieval of information, *see* Search operation
Return address, removal, 27
RETURN command, 12, 27, 37, 39, 45, 48, 70
  bytes used, 28
  use, 28
Reverse characters, typing of, 6
Reverse video, 6, 37
RND command, use, 41, 42
ROM, 51
Routine(s), 18
  lay out, 14
RS-232C port, 47, 53
RS-232C serial buss, 50
Rules, set of, *see* Protocols
RUN command, 14, 42, 43, 73, 74, 82
"Run-time" question, 15

s$ string, 71
SA, 49
SAVE command, 58, 71
SCRATCH command, 58
Screen
  checking of, 10
  device number, 47
  dimensions, 5, 6
  display on, 72
  output to, 48
Screen memory, 31
Search operations, 63–65
Secondary addresses, 49
Sequential files, 55, 57, 98
  appending to, 57
  features, 55–56
  for inventory control, 106
  use, 56, 57
Serial busses, disk drive for, 54
Shifted space, use, 38
Shorthand commands, 28
Single-character input, 9
Single disk drive(s), 54
Single entry, 17–18
Single exit, 17
Software, 2, 53
  advanced, *see* Advanced software
  for catalogue, 65
  costs, 104
  maintenance, 111, 112
  for multiple data file coordination, 64
  objective of, 3
  professional, 3
  quality, 1
  utility, 14
Software packages, standard, 114
Sort key(s), 64
Sort operation, 63–65
  characteristics for, 64
  designing of file for, 64–65
Space(s), 27, 29
  saving of, 27
  wasting of, 27
"SPC(," 28
Special characters, spelling out of, 67
Speed, 14, 27
  disk files and, 57
  execution of, 16
  loss of, 16, 38
  requirements, utility, 13
  see also Time
Spreadsheet(s), 114
Spurious "variable," 75–76
Stack
  cleaning of, 39
  popping of, 39
STEP defaults, size of, 38
STOP command, use, 41, 42
Storage
  brute force, 106
  saving of, 29
STR$ function, 28
  use, 35
String(s)
  conversion of floating-point number to, 35
  operation on, 35
String names, scanning of, 74
String variable, 30–31
  value, assigning of, 31
Structured programming, 17–19
Structuring of system, 102–104
Subroutine(s)
  exiting from, 12
  high-speed, 18
Subroutine call, 27
SUPERLIST program, 2, 7, 21, 67–68
  anomalies, 75–76
  DOMINOES program, 90–96
  initialization, 68–70
  printing the reference, 74–75
  processing a line, 72–74
  starting of, 71–72
  SUPERLISTing, 76–81
  SYS CHECKSUM, 81–85
  TURTLEWALK program, 85–90
  utilities, 70–71
SUPERLISTing, 52, 76–81

SuperPet, 4, 6
Support program, 25
Symbols, 33
"Syntax error," 26
  in SUPERLIST program, 73
SYS CHECKSUM program, 40, 81–84
SYS routine, 41
System
  expansion of, 114–115
  maintenance of, 111–114
  structuring of, 102–104

Tandy machines, 1, 2
Tape, 55
Telephone (modem) communications, 103
Terminal, 1
Test case, 22
Test plan, 22, 101, 110, 111
  draft, 22
Test procedure, 22
Testing, 111
  of program, 21–23
Text, reading of, 3
THEN, 45, 72, 73, 74
TI variable, 30
Time
  balance with memory, 26
  loss of, 34
  saving of, 26, 27, 29, 31, 35
  see also Speed
Timing
  on mainframes, 17
  on microcomputers, 17
  requirements, 22
  RS-232C buss and, 50
TO command, function, 75
Token, tokenizing, 28, 98
  features, 59–61
Transaction(s), 102
  collection of, 106
Trouble report, 112
True statement, 43, 44
TURTLEWALK program, 85–90

Unary functions, 44
  numeric, DEFining of, 43
Uppercase letters, ASCII conversion for, 50, 51
User, 3–4
  class of, 9
  writing for, *see* Writing, for user
User-defined functions, 44
User Manual, 19, 20, 21, 22, 101, 111, 112
  draft, 22, 109
  function, 20
  generation, 10–20
  review, 109–110
USR function, in SYS CHECKSUM program, 83
USR routine, 41
Utility(ies), 11, 12–13
  bytes for, 25
  high-speed, 15
  low-speed, 15
  required, 11, 13
  software, 13
  SUPERLIST, 70–71
Utility code, 13
Utility routines, 110

VAL function, 35
  use, 75
Value(s), 32
  of variable, 29, 30, 31
Variable(s)
  array, 75
  beginning of, 26
  clearing of, 42
  cross referencing of, 67, 68
  defining of, 32
  handling of, in SUPERLIST program, 72
  spurious, 75–76
  types and use, 29f, 29–31
Variable flag (vf), 73, 74
Variable line spacing, 49
Variable name, 29, 30, 31, 32
vf, 73, 74
VIC, poke, 6
VIC 20, 1, 50
  advanced software for, 7
Video, reverse, 6, 37

WAIT command, use, 41–42
Warehouse program, 103, 104
  architecture, 106
  design for, 106–107
  use of, 107–108
Warehouse station, 104
  for Inventory Control System, 105
Warehousing function, inventory control system for, 105
WEDGE program, 41
Whistles, 37

''Window,'' 6, 7
Word(s), transformation into
 token, 28
Word Machine, 2, 65, 115
 function, 3
Word processor, 114
Wordcraft, 3
WordPro, 3, 114, 115
Work string (x$), 74
Wrapup, 72, 74
Writing
 advanced software, 97
 for user, 9-11
  program documentation, 19-21
  program organization, 14-17
  program planning, 11-14
  structured programming, 17-19
  testing, 21-23

x$, 74
X variable, 30
X1 variable, 30
XI variable, 30
XOR function, 41, 69, 71, 81
XOR operation, performance, in
 CHECKSUM program, 82
XORing, 72

Advanced BASIC Programming for the Commodore 64 and Other Commodore Computers

Michael Richter

Now—an advanced BASIC programming guide for experienced Commodore users! This professional-level text shows you how serious software programs are written .... how to read and use them .... how to know a good program when you see one ... and how to gain knowledge through writing programs yourself. Covers techniques for the organization, design, development, and marketing of advanced BASIC software—comprehensively citing underlying principles and example programs along the way to give you a practical as well as theoretical approach to learning on your Commodore.

In this book you will find complete, concise sections on:

- the principles of BASIC
- interoperability
- writing for the user
- the mechanics of a program
- bits, bytes, characters, and numbers
- coding tricks
- using files
- SUPERLIST
- pursuing a project from the first customer call to a finished system

CONTENTS

Introduction / Writing for the User / Mechanics of a Program / Devices / SUPERLIST—an Example & a Tool / Pursuing a Project

ISBN 0-89303-302-2